USER MANUAL

Accessory 9PN
Pcomm32

PMAC 32-Bit Driver

3xx-W32DRV-xUxx

October 30, 2003

DELTA TAU
Data Systems, Inc.
NEW IDEAS IN MOTION ...

Single Source Machine Control *Power // Flexibility // Ease of Use*
21314 Lassen Street Chatsworth, CA 91311 // Tel. (818) 998-2095 Fax. (818) 998-7807 // www.deltatau.com

Copyright Information

© 2003 Delta Tau Data Systems, Inc. All rights reserved.

This document is furnished for the customers of Delta Tau Data Systems, Inc. Other uses are unauthorized without written permission of Delta Tau Data Systems, Inc. Information contained in this manual may be updated from time-to-time due to product improvements, etc., and may not conform in every respect to former issues.

To report errors or inconsistencies, call or email:

Delta Tau Data Systems, Inc. Technical Support
Phone: (818) 717-5656
Fax: (818) 998-7807
Email: support@deltatau.com
Website: http://www.deltatau.com

Operating Conditions

All Delta Tau Data Systems, Inc. motion controller products, accessories, and amplifiers contain static sensitive components that can be damaged by incorrect handling. When installing or handling Delta Tau Data Systems, Inc. products, avoid contact with highly insulated materials. Only qualified personnel should be allowed to handle this equipment.

In the case of industrial applications, we expect our products to be protected from hazardous or conductive materials and/or environments that could cause harm to the controller by damaging components or causing electrical shorts. When our products are used in an industrial environment, install them into an industrial electrical cabinet or industrial PC to protect them from excessive or corrosive moisture, abnormal ambient temperatures, and conductive materials. If Delta Tau Data Systems, Inc. products are directly exposed to hazardous or conductive materials and/or environments, we cannot guarantee their operation.

Table of Contents

USERS MANUAL 1
Introduction 1
Document Layout 1
A Global View of the Library 2

GETTING STARTED 3
Installation 3
If the installation fails 3
Windows NT 3
Windows 95/98 3
Usage of PComm32 4
Linking Your Code with the Driver 4
Initializing Communication 5
Communication Modes 5
Shutting Down Communication 7
A Quick Guide to ASCII Communication Functions 7
UNICODE Versus ANSI Compiles 8
Error Handling 8

PCOMM32 FEATURES 9
Using Interrupts 9
Send A Message To A Window 9
Call A Function 9
Start A Thread Function 9
Set An Event 10
Downloading To PMAC 10
Downloading ASCII PMAC Data 10
Multiple File Cooperative Downloading 10

PCOMM32 DPR FEATURES 11
A Global View of the DPR Support Functions 11
Control Panel 11
Fixed Real Time Data Buffer 11
Fixed Background Data Buffer 11
Variable Background Data Buffer 11
Binary Rotary Buffer 12
Read/Write Functions 12
Initialization /Shutdown 12
Memory Organization/Correct Order of Initialization 12
Using the DPR Control Panel 14
Action Bits 14
Request Bits 14
Assigning A New Feedrate Override 15
Feedrate Override Enable Bits 15
Using the DPR Real Time Fixed Data Buffer 16
Using the DPR Background Fixed Data Buffer 17
Using the DPR Variable Background Read/Write Data Buffer 19
Variable Read Data Buffer 19
Variable Write Data Buffer 22
Using the DPR Binary Rotary Motion Program Buffer 22
PmacDPRRotBufInit() 23
PmacDPRRotBufRemove() 23
PmacDPRRotBufChange() 23
PmacDPRRotBufClr() 23

PmacDPRRotBuf() *23*
PmacDPRBufLast() *23*
Downloading "Stuffing" Functions *24*

PROGRAMMERS REFERENCE **27**
Programmers Reference (Introduction) 27
Important Information About Method of Communication Being Used by PComm32 ..... *27*
Maximum Number of Response Buffer Characters *27*
UNICODE Versus ANSI Compiles *27*

INITIALIZATION, SHUTDOWN AND CONFIGURATION FUNCTIONS **29**
OpenPmacDevice() 29
ClosePmacDevice() 29
PmacAbortDPRTest() 29
PmacConfigure() 30
PmacGetDpramAvailable() 30
PmacInBootStrapMode() 30
PmacDPRTest() 30
PmacSERIsOpen() 31
PmacSERGetBaudrate() 31
PmacSERGetHandle() 32
PmacSERGetPort() 32
PmacSEROnLine() 32
PmacSERSetBaudrate() 32
PmacSERSetComm() 33
PmacSERSetPort() 33

ASCII COMMUNICATION FUNCTIONS **35**
PmacFlush() 35
PmacGetAsciiComm() 35
PmacGetBuffer*X*() 35
PmacGetControlResponse*X*() 36
PmacGetLine*X*() 36
PmacGetResponse*X*() 37
PmacReadReady() 37
PmacSendChar*X*() 38
PmacSendCommand*X*() 38
PmacSendLine*X*() 38
PmacSetAsciiComm() 38

DOWNLOADING FUNCTIONS **41**
PmacAbortDownload() 41
PmacAddDownloadFile*X*() 41
PmacCompilePLCC*X*() 41
PmacDownload() 42
PmacDownloadFile() 42
PmacDownloadFirmwareFile() 42
PmacMultiDownload() 43
PmacRemoveDownloadFile*X*() 44
PmacSetMaxDownloadErrors() 45
PmacThreadDownload*X*() 45

DUAL PORTED RAM CONTROL PANEL **47**
PmacDPRControlPanel() 47
PmacDPRGet*Action*Bit() and PmacDPRSet*Action*Bit() 47
PmacDPRGetFOEnableBit() and PmacDPRSetFOEnableBit() 48
PmacDPRGetFOValue() and PmacDPRSetFOValue() 48

PmacDPRGetRequestBit() and PmacDPRSetRequestBit() 49

DPR REAL TIME FIXED DATA BUFFER 51
DPR Real Time Fixed Data Buffer Initialization and Handshaking 51
DPR Real Time Fixed Data Buffer Query Routines 51
Global 51
Coordinate System 52
Motor 52

DPR BACKGROUND FIXED DATA BUFFER 55
DPR Background Fixed Data Buffer Initialization and Handshaking 55
DPR Background Fixed Data Buffer Query Routines 56
Data Query Functions 56

DPR VARIABLE BACKGROUND READ/WRITE DATA BUFFER 59
PmacDPRBackground() 59
PmacDPRBufLast() 59
PmacDPRGetVBGAddress() 60
PmacDPRGetVBGNumEntries() 60
PmacDPRGetVBGDataOffset() 60
PmacDPRGetVBGAddrOffset() 60
PmacDPRGetVBGServoTimer() 61
PmacDPRGetVBGStartAddr() 61
PmacDPRGetVBGTotalEntries() 61
PmacDPRVarBufChange() 61
PmacDPRVarBufInit() 62
PmacDPRVarBufRead() 62
PmacDPRVarBufRemove() 63
PmacDPRWriteBuffer() 63

DUAL PORTED RAM BINARY ROTARY BUFFER FUNCTIONS 65
PmacDPRAsciiFileToRot() 65
PmacDPRAsciiStrToBinaryFile() 65
PmacDPRAsciiStrToRot() 66
PmacDPRAsciiToBinaryFile() 67
PmacDPRBinaryFileToRot() 67
PmacDPRBinaryToRot() 68
PmacDPRRotBufInit() 69
Non-Parameter Inputs 69
PmacDPRRotBufRemove() 70
PmacDPRRotBufChange() 70
PmacDPRRotBufClr() 70
PmacDPRRotBuf() 71

DPR NUMERIC READ AND WRITE 73
General Information 73
Standard Read/Write 73
Masking 73
Dual Word Conversion 73
PmacDPRDWordBitSet() 74
PmacDPRFloat() 74
PmacDPRGetDWord() 75
PmacDPRGetDWordMask() 75
PmacDPRGetFloat() 75
PmacDPRGetMem() 75
PmacDPRGetWord() 76
PmacDPRResetDWordBit() 76

PmacDPRSetDWord() 76
PmacDPRSetDWordBit() 76
PmacDPRSetDWordMask() 76
PmacDPRSetFloat() 77
PmacDPRSetMem() 77
PmacDPRSetWord() 78

INTERRUPT FUNCTIONS 79
General Information Interrupt Functions 79
PmacINTRFireEventInit() 79
PmacINTRFuncCallInit() 80
PmacINTRRunThreadInit() 80
PmacINTRTerminate() 80
PmacINTRWndMsgInit() 80
Serial Checksum Communication Functions 81
PmacSERDoChecksums() 81
PmacSERCheckResponseA() 81
PmacSERCheckSendLineA() 82
PmacSERCheckGetLineA() 82
PmacSERCheckResponse() Source/Example 82

VARIABLE FUNCTIONS 85
PmacGetVariable() 85
PmacSetVariable() 85
PmacSetIVariable() 85
PmacGetIVariable() 86

UTILITY FUNCTIONS 87
PmacGetRomDateA() 87
PmacGetRomVersionA() 87
PmacGetUserHandle() 87
PmacGetPmacType() 87
PmacGetIORegister() 87
PmacSetIORegister() 88
getBitValue() 88
hex_long2() 89

DATA TYPES, STRUCTURES, CALLBACKS, AND CONSTANTS 91
DOWNLOADGETPROC 91
DOWNLOADPROGRESS 91
DOWNLOADMSGPROC 91
DPRTESTMSGPROC 91
DPRTESTPROGRESS 92
Dpr_Bin_Rot_Errors 92
GLOBALSTATUS 93
MOTION 93
MOTIONMODE 94
PROGRAM 94
SERVOSTATUS 94

EXTENDED FUNCTIONS 95
PmacGetLineEx*X*() 95
PmacGetResponseEx*X*() 96
PmacGetControlResponseEx*X*() 96
PmacDPRRealTimeEx 97
PmacDPRBackgroundEx() 98
PmacDPRVarBufInitEx() 98

PmacDPRVarBufChangeEx() 99
PmacDPRWriteBufferEx() 99

COMMUNICATION APPLICATION NOTES 103
Common Communication Traps 103
Suggested Startup Procedure for BUS 104
Serial Communications 104
Slow Serial Communication 104
Less Common Communication Problems 105
Rare Communication Problems 106
Common Dual Ported Ram Traps 106
DOS/Windows 3.x Users 107
Windows 95/98 & NT 4.0 Users 107
Rare Problems 108
Terminal Example Program 108

INDEX 110

USERS MANUAL

Introduction

The Delta Tau PComm32 Communication Driver is a set of more than 250 functions written as a development tool for the creation of PMAC 32-bit applications on Windows 95/98 or Windows NT. Nearly all methods of communication to PMAC are included. PComm32 may be used by the PMAC1 as well as the PMAC2. The routines were designed with robustness, speed and portability in mind.

Note:

Methods of communication include the bus, serial, and Dual-Ported Ram.

PComm32 has been tailored to be compatible with Borland and Microsoft development products using C, C++, Visual Basic or Delphi. The manual assumes that you know the basics of working in Windows and general programming practices.

Document Layout

The complete manual is divided into two major sections, the User's Manual and the Programmer's Reference. The user's manual describes fundamental use whereas the programmer's reference details all of the PComm32 Driver Library functions. The user's manual outline is described below:

Chapter 1 - Overview: Describes the purpose structure and use of the manual, the layout of the manual, and the compatibility of the PComm32 features.

Chapter 2 - Getting Started: Covers installing the software for various Windows applications, establishing initial communications with PMAC and troubleshooting any communications problems that arise.

Chapter 3 - PComm32 Features: Describes how to use interrupts, multiple file downloading.

Chapter 4 - Dual Ported RAM Features: Provides information on using the dual ported RAM support functions of the PComm32 Library.

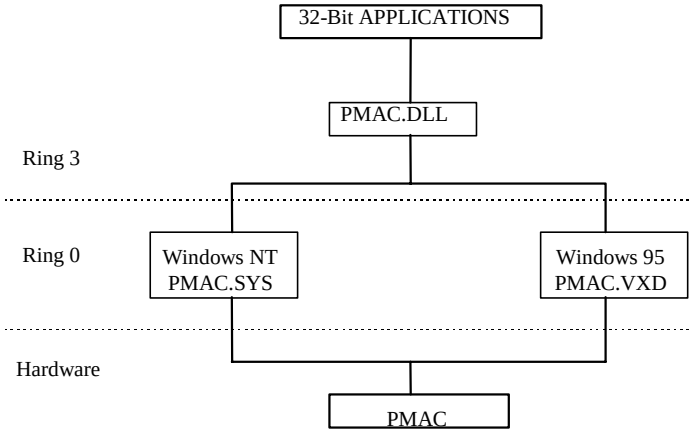
A Global View of the Library

PComm32 can be used for Windows 95/98 and Windows NT application development. The library is structured such that an application using the library created for the Windows 95/98 will also be able to run under Windows NT as long as the application itself uses no operating system specific functions.

PComm32 itself consists of three files.

- PMAC.DLL - A 32-bit DLL.
- PMAC.SYS - A Windows NT kernel driver.
- PMAC.VXD - A Windows 95/98 32-bit virtual device driver.

The illustration below shows how these modules are related.



GETTING STARTED

Installation

Run the "SETUP.EXE" application on Disk #1 of the distribution. This will place the driver files in the appropriate locations on your system. If you are in Windows NT reboot your system. To test if installation went well, run the PMACTEST.EXE application.

Starting from this installation "Version 10.32" and later revisions a Delta Tau Shared folder will be created in the system common files folder. Delta Tau shared applications will be saved in this folder. At this point MotionExe is the only such file. More files will be installed in this folder in later revisions.

If the installation fails

Make sure of all jumpers and settings on the PMAC card match the parameters you have selected in the configuration dialog box.

Windows NT

Most communication problems will be reported in the Windows NT event logging system. Using the event viewer application located in the administration tools group you can view any error reports the driver made during communication with PMAC.

After installation you can manually check that the driver is installed successfully. This can be done in several ways.

1. Open the Control Panel "Devices" list and see if in fact the PMAC device appears in the list of installed devices. NT only. Further, make sure that PMAC device status show as started in this list.
2. Go to administrative tools and run NT Diagnostics. Look under the Devices or Drivers section for PMAC.
3. Make sure that PMAC key is properly set in the registry. In Windows NT it should be
    a. \HKLM\SYSTEM\CurrentControlSet\Services\PMAC is of the form
    b. ImagePath ="\SystemRoot\System32\drivers\Pmac.SYS"
    c. etc.

Windows 95/98

The Windows 95/98 virtual device driver (pmac.vxd) is a dynamically loading driver. During installation this file is copied to the Windows system directory.

In order for the library file (pmac.dll) to find and load pmac.vxd there has to be a symbolic name available for the path location of pmac.vxd. In order to accomplish this, symbolic name information is written to the registry at location:

Run regedit.exe and check the CurrentControlSet parameters under for the PMAC configuration keys.

In Windows 95/98 you should have:

HKLM\System\CurrentControlSet\Control\SessionManager\KnownVxDs

in the form of:

PMAC0 = C:\Windows\System\pmac.vxd

PMAC1 = C:\Windows\System\pmac.vxd, etc.

Usage of PComm32

In this section we discuss the usage of PComm32 in general, as well as specifically through an example program written in Microsoft Visual C++ (MSVC).

The example MSVC program called PMACTEST uses Microsoft Foundation Classes (MFC). The important details of linking with the library, as well as initializing and shutting down communication are discussed in this section. The PMACTEST project is located in the "<Install Directory>\PmacTest\" directory.

More test programs and illustrative examples using C++ Builder 4.0 and VB 6.0 are provided for programmers' reference. These projects are located in their respective folders under "<Install Directory>\PmacTest\" folder.

Linking Your Code with the Driver

For any one operating system PComm32 will use two modules, a DLL (pmac.dll) and a kernel driver (pmac.vxd for 95/98, pmac.sys for NT). Your application must interface with the DLL to communicate to the kernel driver (which in turn talks to the PMAC directly). Methods for interfacing to DLL's for any given development environment are well documented so all the details won't be discussed here. At a minimum, what is needed to interface to a DLL is the function prototypes and the location of the DLL in the operating system. The prototypes for the PMAC.DLL functions are detailed in the reference manual, as well as the header files that come with the library. The installation program of the driver places the pmac.dll in the system directory.

The PMACTEST program uses the LoadLibrary() and GetProcAddress() system API calls to link with the pmac.dll. In the RUNTIME.CPP source file you will find a routine, RuntimeLink() which performs these tasks. A small excerpt from the file is show below:

```
#include "runtime.h"
HINSTANCE RuntimeLink()
{
  // Get handle to PMAC.DLL
  hPmacLib = LoadLibrary(DRIVERNAME);

// Get some DLL procedure addresses,
//  for example OpenPmacDevice and ClosePmacDevice
                                DeviceOpen =
                                (OPENPMACDEVICE)GetProcAddress(hPmacLib,"OpenPmacDevice");
                                DeviceClose =
                                (CLOSEPMACDEVICE)GetProcAddress(hPmacLib,"ClosePmacDevice");
  .
  .
// Check validity of the procedure addresses
  if(!DeviceOpen || !DeviceClose ||){
        // Report an error, couldn't link with one or more functions
        return NULL
  }
  else
    return hPmacLib;
}
```

The header file, "runtime.h" has the type definitions for the function pointers used in the code above. A portion of the file is shown below:

```
#define DRIVERNAME TEXT("Pmac.dll") //name of the user-dll driver

//*************************************************************
// COMM Type Defines
//*************************************************************
typedef BOOL (CALLBACK* OPENPMACDEVICE)(DWORD dwDevice);
typedef BOOL (CALLBACK* CLOSEPMACDEVICE)(DWORD dwDevice);
.
.
```

Initializing Communication

In the demo program PMACTEST a class has been created, CPmacTestDoc (see pmactestdoc.cpp), which has been designed to handle all communication between the application and PMAC. It establishes communication by a member function called InitDocument(). InitDocument() is shown below:

```
BOOL CPmacTestDoc::InitDocument(LPCTSTR lpszPathName)
{
  TCHAR str[_MAX_PATH], vs[30],ds[30];

  // Get handle to PMAC.DLL, and get procedure addresses
  if(!m_hPmacLib)
     m_hPmacLib = RuntimeLink();

  if(m_hPmacLib == NULL)
     return FALSE;// We failed to load the library or functions

  if(m_bDriverOpen)// Only call OpenPmacDevice() once
     return TRUE;
// Call OpenPmacDevice()
  m_bDriverOpen = DeviceOpen(m_dwDevice);

  if(m_bDriverOpen) {
     DeviceGetRomVersion(m_dwDevice,vs,30);
     DeviceGetRomDate(m_dwDevice,ds,30);
     sprintf(str,TEXT("V%s %s"),vs,ds);
     SetTitle(str);
     return TRUE;
  }
  else {
     AfxMessageBox("Could not initialize PMAC Comm.");
     SetTitle(TEXT("Not Linked.."));
     return FALSE;
  }
}
```

Notice the two steps to initializing communication to a PMAC:

1. Load the DLL, and get procedure addresses
2. Call OpenPmacDevice() with the PMAC device number

The PMAC device number is 0 for the first PMAC in your system, 1 for the second and so on.

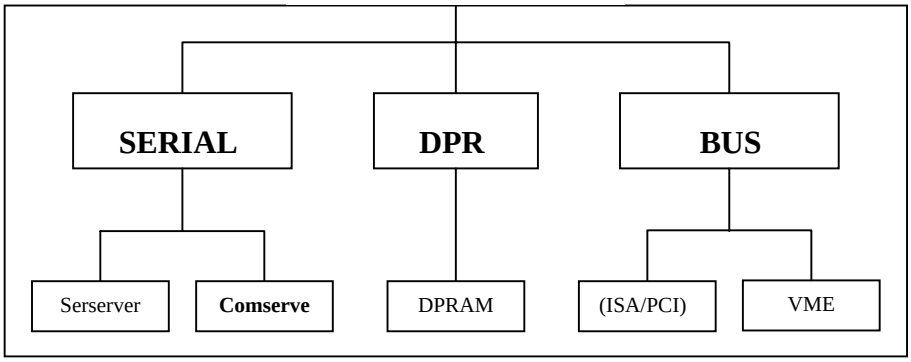
Once PMAC communication is initialized, all other functions in the library may be called.

Communication Modes

There are three methods by which PComm32 may communicate to PMAC, over the Bus (ISA/PCI/VME), Dual Ported Ram, or the Serial Port.

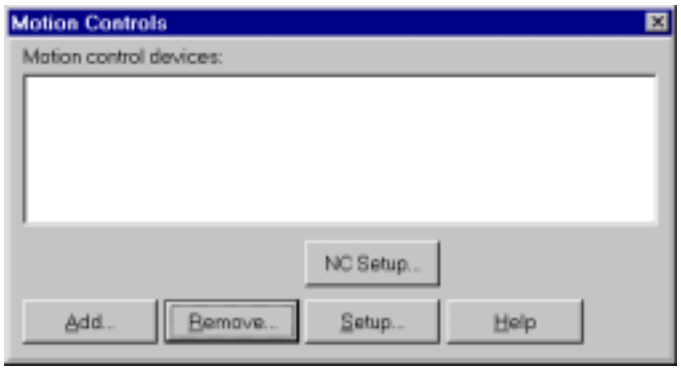
Starting from this enhanced version of Pcomm32 a faster and transparent serial communication program “Comserver” has been introduced. Comserver is supported under Windows 98 and NT 4.0. Under Windows 95, serial communication still uses the conventional Serserver. This change in no way should affect your development environment. The following diagram illustrates the structure of PMAC.DLL communication paths.

Communication Modes



Immediately after initialization (after a call to OpenPmacDevice()) the method used depends on what is stored in the system registry (either BUS or SERIAL). To change the method used at startup call the PmacConfigure() function so that the change is saved in the registry. Delta Tau's PMAC driver configuration program MotionExe is meant to update the registry for each PMAC. Launch MotionExe from the program group Pcomm32 or any other Delta Tau's application. Following Motion control applet appears.

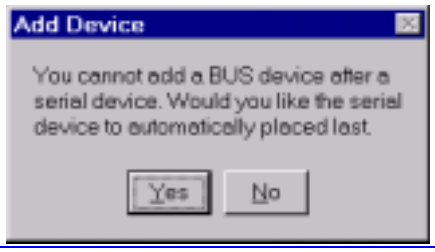
Add a new device. (The first device number is always 0.)



After selecting the appropriate communication mode press OK to update the registry. A message of successful loading of PMAC appears.

Note:

If for any given PMAC device both BUS and Serial communications have to be established BUS needs to be at lower device number than the Serial. If you want to add a PMAC device on the BUS after configuring the Serial device following message will appear.


Add Device
You cannot add a BUS device after a serial device. Would you like the serial device to automatically placed last.
Yes
No

Consequently MotionExe will rearrange the devices in the list in the desired order. Removing a device from the middle of the list will also print a “discontinuous devices” warning message and rearrange the remaining devices in the continuous fashion.

Shutting Down Communication

To release any resources the driver has allocated it is good programming practice to shutdown any communication links that have been opened. In PMACTEST, the demo application, this is done within the CPmacTestDoc class as shown below:

```
void CPmacTestDoc::CloseDocument()
{
  if(m_bDriverOpen) {
     // Call ClosePmacDevice()
      m_bDriverOpen = !DeviceClose(m_dwDevice);
  }
  if(m_hPmacLib) {// Free the library
    FreeLibrary(m_hPmacLib);
    m_hPmacLib = NULL;
  }
}
```

Notice the two steps in shutting down communication:

1. Shut down the PComm32 driver with a call to ClosePmacDevice()
2. Call FreeLibrary so that the operating system may unload the PMAC.DLL from memory

A Quick Guide to ASCII Communication Functions

The PComm32 supports both UNICODE and ANSI compiles. For a small portion of PMAC.DLL there are two sets of nearly identical functions, one for UNICODE and one for ANSI. The manual refers to the identical functions by using an "*X*" suffix. For example both the PmacGetResponseW() and PmacGetResponseA() functions are referred to in the manual as PmacGetResponse*X*().

With so many ASCII functions to choose from, a few words on which functions should be the most popular is appropriate.

Most if not all of your communication with the PMAC can be handled with the PmacGetResponse*X*() function. This function will send a command string (i.e. “#1j+, “?”, “Open Prog1”, etc.) to the PMAC and retrieve and place any pending responses within a response buffer for your use. This is an efficient and *safe* function to use. The word *safe* is emphasized because there are functions (i.e. PmacSendChar*X*(), PmacGetLine*X*() and PmacSendLine*X*()) which, if not used carefully can cause un-synchronized communication (the mismatching of PMAC Commands to PMAC responses). PmacGetResponse*X*() always matches the command string with the response string or else it “times out”.

For getting responses to a PMAC control-character command it's easiest to use PmacGetControlResponse*X*().

UNICODE Versus ANSI Compiles

For those folks programming in C/C++ (or any other program that supports include files and a macro preprocessor) you may keep your PComm32 specific code independent of the standard being used by using the definitions below (See UTIL.H).

```
#ifdef UNICODE

#define PmacGetResponse PmacGetResponseW
#define PmacGetControlResponse PmacGetControlResponseW
#define PmacSendChar PmacSendCharW
#define PmacSendLine PmacSendLineW
#define PmacGetLine PmacGetLineW
#define PmacGetBuffer PmacGetBufferW
#define PmacSendCommand PmacSendCommandW
#define PmacMultiDownload PmacMultiDownloadW
#define PmacAddDownloadFile PmacAddDownloadFileW
#define PmacRemoveDownloadFile PmacRemoveDownloadFileW
#define PmacRenumberFiles PmacRenumberFilesW
#define PmacGetErrorStr PmacGetErrorStrW
// Extended functions
#define PmacGetLineEx PmacGetLineExW
                      #define PmacGetResponseEx PmacGetResponseExW
                      #define PmacGetControlResponseEx
                                 PmacGetControlResponseExW
#else

#define PmacGetResponse PmacGetResponseA
#define PmacGetControlResponse PmacGetControlResponseA
#define PmacSendChar PmacSendCharA
#define PmacSendLine PmacSendLineA
#define PmacGetLine PmacGetLineA
#define PmacGetBuffer PmacGetBufferA
#define PmacSendCommand PmacSendCommandA
#define PmacMultiDownload PmacMultiDownloadA
#define PmacAddDownloadFile PmacAddDownloadFileA
#define PmacRemoveDownloadFile PmacRemoveDownloadFileA
#define PmacRenumberFiles PmacRenumberFilesA
#define PmacGetErrorStr PmacGetErrorStrA

// Extended functions
#define PmacGetLineEx PmacGetLineExA
#define PmacGetResponseEx PmacGetResponseExA
                      #define PmacGetControlResponseEx
                                 PmacGetControlResponseExA
#endif // !UNICODE
```

Error Handling

Error handling will be implemented in the next release of the driver. Any part of this documentation which implies that an error code or string exists is only for preparation of these items.

PmacGetError() will return 0 if all went well. The error string does not get reset after every call to GetResponse() whereas the Error number does.

PCOMM32 FEATURES

Using Interrupts

Interrupts are provided for all Windows 95/98 and Windows NT operating systems. There are four methods of interrupt notification for your program:

1. Send a message to a window.
2. Call a function.
3. Start a thread function.
4. Set an event.

All four methods have a separate initialization function. Common to all initialization functions is the parameter *ulMask*. This parameter determines the interrupt service vector(s) to be used for the interrupt initiated by the function.

The least significant byte of *ulMask* controls which conditions will generate an interrupt. A bit value of 0 enables, 1 disables.

Bit	PMAC Signal
0	In Position of Coordinate System
1	Buffer Request (PMAC's request for more moves)
2	Error, A motor(s) in the coordinate system has had a fatal following error
3	Warning, A motor(s) in the coordinate system has had a warning following error
4	Host Request, PMAC has an ASCII response for the host
5-7	User programmable, see PMAC User's Guide, Writing a Host Communications Program

The PmacINTRTerminate() function is provided to shutdown the interrupt service.

Send A Message To A Window

This interrupt method will use the Windows SendMessage() function to send a message to a window handle (HWND). Both the message and the windows handle are provided by the programmer. Use the function:

```
BOOL  CALLBACK PmacINTRWndMsgInit(DWORD dwDevice, HWND hWnd, UINT msg,
ULONG ulMask);
```

to initiate this method.

Call A Function

This interrupt method will cause a function provided by the programmer to be called. Use the function:

```
BOOL  CALLBACK PmacINTRFuncCallInit(DWORD dwDevice, PMACINTRPROC
pFunc, DWORD msg, ULONG ulMask);
```

to initiate this method.

Start A Thread Function

This interrupt method will start a thread function provided by the programmer. It is the programmer's responsibility to terminate the thread. Use the function:

```
BOOL  CALLBACK PmacINTRRunThreadInit(DWORD dwDevice,
LPTHREAD_START_ROUTINE pFunc, UINT msg, ULONG ulMask);
```

to initiate this method.

Set An Event

This interrupt method will set an event provided by the programmer. It is the programmer's responsibility to reset the event. Use the function:

```
BOOL  CALLBACK PmacINTRFireEventInit(DWORD dwDevice,HANDLE
hEvent,ULONG ulMask);
```

to initiate this method.

Downloading To PMAC

Downloading ASCII PMAC Data

Downloading of PMAC motion, plc, configuration files etc. may easily be done by the use of the PmacDownload() or PmacDownloadFile() functions. PmacDownloadFile(), the simplest of all the download functions, takes a *pure* native PMAC language ASCII file and downloads it straight to the card, no bells or whistles. PmacDownload(), can do this also in addition to:

1. Parse Macros (i.e. #define, #include etc.)
2. Compile PLC's to PLCC's
3. Create a map file from macros
4. Create a log file of download progress
5. Invoke a callback function for displaying progress (same text as the log file that can be created)
6. Download a file or a buffer through a line retrieval call back function
7. Update a progress bar through a call back function

Multiple File Cooperative Downloading

A problem exists when downloading PLCC programs to the PMAC from multiple applications. All PLCC programs that should be running within PMAC should be downloaded at the same time (in other words, all in one file). When application "A" creates and downloads a file with PLCCs, it has no way of knowing that other applications in the system also require that their PLCC bearing files need to be downloaded as well.

The solution is to have a central location for which all applications in the system register the set of files they use. From this registered set, any one of the cooperating applications can download all plcc programs required by all cooperating applications. Download procedures have been developed to utilize this solution and eliminate any problems associated with PMAC compiled PLCs and PLCCs.

To remain compatible with any products that Delta Tau and Third party vendors create, you must implement the following set of functions, instead of the standard Download() procedures:

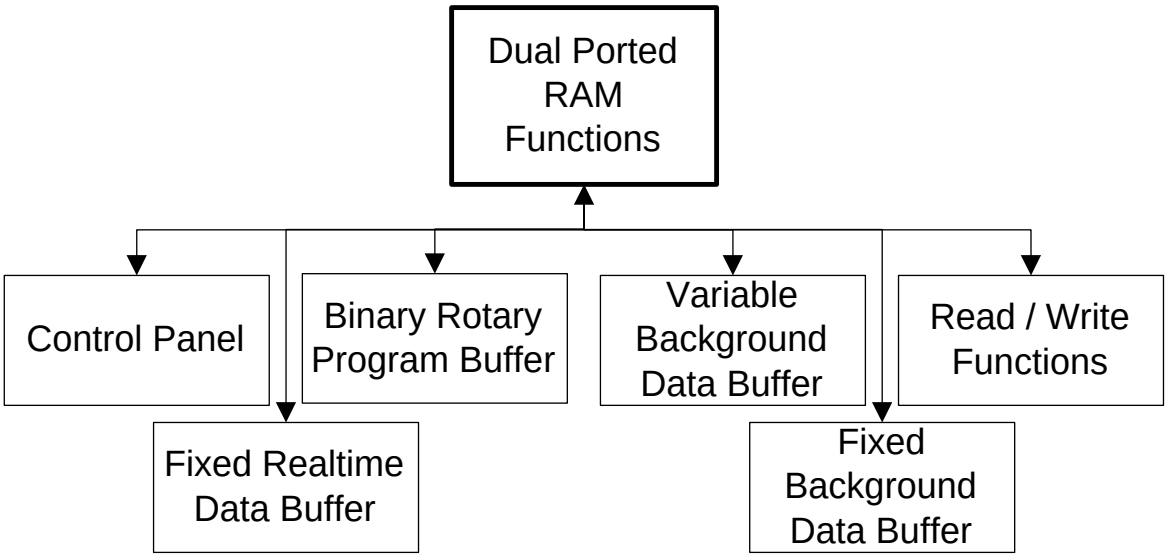
```
PmacAddDownloadFileX()
PmacMultiDownloadX()
PmacRemoveDownloadFileX()
```

The PmacAddDownloadFile*X*() appends to a centrally located list of files to be downloaded every time PmacMultiDownload*X*() is called to download files to PMAC. PmacRemoveDownloadFile*X*() will remove the files from the list.

PCOMM32 DPR FEATURES

A Global View of the DPR Support Functions

The majority of the functions in this library are Dual Ported RAM (DPR) support functions. Their descriptions in the reference manual have been grouped by functionality as shown below.



Control Panel

The PMAC provides an area in dual ported RAM that emulates the PMAC's hardware control panel *connector*. When enabled, functions are available to:

1. Jog+, Jog-, Pre-Jog , Home (Motors only).
2. Run, Stop, Step and Feed Hold (Coordinate System only)
3. Assign a feedrate override value for a particular coordinate system.

Fixed Real Time Data Buffer

PMAC has an automatic DPR feature, the Fixed Real Time Data Buffer, in which PMAC continually updates a specific area of DPR with a fixed data structure. This data structure is full of meaningful motor information and can be by a host application to show positions, velocities etc. in real time. The data in this feature gets updated in PMAC's Real Time Interrupt.

Fixed Background Data Buffer

This automatic DPR feature is very similar to the Fixed Real Time Data Buffer in that a fixed data structure is copied by PMAC into a specific area of the DPR. The difference is that the information is coordinate system specific. Furthermore the information is updated in PMAC's background cycle.

Variable Background Data Buffer

PMAC has the ability to copy up to 128 of it's own registers to the DPR for the host to use. The user may specify the location of these 128 registers in PMAC's memory map (hence the name Variable Background Data Buffer).

Binary Rotary Buffer

This PMAC DPR automatic feature can be used to efficiently download large part programs to PMAC's internal rotary buffer. The routines included to support this feature convert the PMAC ASCII to PMAC Binary before placing the code in the DPR for PMAC to retrieve.

Read/Write Functions

Numeric transfers of 16 and 32 bit wide numbers may be read or written two with this set of PComm32 routines. Floating-point values are supported for 32 bit transfers. In addition, several helper routines exist for setting individual bits of DPR memory.

Initialization/Shutdown

Once initialization of communication has been successfully completed, all other DPR functions may be called (of course, only if DPR is present in the system). A complete description of their use and examples are provided in the reference manual.

There is a distinction between "initializing" and "turning on" a DPR feature that must be discussed. Some DPR features require two actions to be taken before they are "running" (like an electrical appliance in your home requires that the circuit breaker as well as the switch on the appliance to both be turn on). First you initialize the feature (i.e. for the DPR Rotary buffer call the *PmacDPRRotBufInit()*). After the feature is initialized the turning off or on of the feature may be done with a yet a different function call (i.e. for the DPR Rotary buffer call the *PmacDPRRotBuf()*).

In addition to initialization and shutdown order being important, the order in which a program turns on the DPR features is also *critical*. What you need to know about the order of turning features on is: If you are using more than one DPR automatic feature, always turn on the DPR Rotary Buffer last. The reason for this is that a call to *PmacDPRRotBuf()* that turns the feature on will open a PMAC program buffer. When a PMAC program buffer is open any attempt to initialize or enable other DPR features will fail (since the driver has to set I-variables to enable a feature, and if a buffer is open the I-variable assignments won't get processed, but rather stored in the buffer).

Memory Organization/Correct Order of Initialization

The order in which the binary rotary buffers and the variable background data buffer are initialized and removed is very important to the successful operation of the buffer(s). This is because they are of variable size, and an assumption is made as to the placement of the memory and the sequence of initialization by the library. The order in which they should be initialized is:

1. Binary rotary buffer #0
2. Binary rotary buffer #1
3. Binary rotary buffers (2-7)
4. Variable background data buffer

The following example code demonstrates this:

```
/* 1. Initialize 1st rotary buffer, rotary #0, coordinate system #1 (always)*/
      if (PmacDPRRotBufInit(dwDevice,0) < 0){
           MessageBox("***DPR Rotary Buffer #1 Error ***\r\n");
           return;
      }
/*2.Initialize 2nd rotary buffer, coordinate system #2 (always)*/
      if (PmacDPRRotBufInit(dwDevice,1) < 0){
           MessageBox("***DPR Rotary Buffer #2 Error ***\r\n");
           return;
      }
// 3. Initialize variable background buffer
      vbg_handle = PmacDPRVarBufInit(dwDevice, 4, VAdarray );
      if ( vbg_handle <0){
           MessageBox("***DPR Background Variable Buffer Error***\r\n");
           return;
      }
// Turn on DPR background data buffer, This MUST BE CALLED
//     BEFORE PmacDPRRotBuf()
      PmacDPRBackground(dwDevice, ON);
// Turn on DPR rotary buffers
      PmacDPRRotBuf(dwDevice, ON);
```

Just like a Last In First Out stack, the order in which the buffers should be removed is:

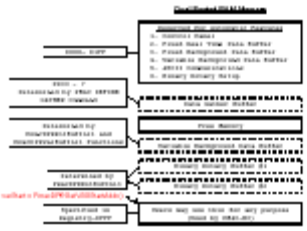
1. Variable background data buffer
2. Binary rotary buffer #7-#2
3. Binary rotary buffer #1
4. Binary rotary buffer #0

The following shut down procedure is generalized to handle any situation:

```
void closeAll(dwDevice){
    //Shut Down
    int buffnum ;
    while( ( buffnum = PmacDPRBufLast(dwDevice) ) > 0 ){
       if( buffnum == 3 )
           PmacDPRVarBufRemove(dwDevice,vbg_handle);//remove BG
Buf
       if( buffnum == 2 )
           PmacDPRRotBufRemove(dwDevice, 1 );//remove BinRotBuf #2
       if( buffnum == 1 )
           PmacDPRRotBufRemove(dwDevice, 0 ); // remove BinRotBuf #1
    }
    if( buffnum < 0 )
       MassageBox("***DPR Buf Remove ERR %d***\r\n", buffnum );
}
```

Here the PmacDPRBufLast() routine is called to determine what type of buffer was placed last in the DPR (3 = Variable Data, 2 = Binary rotary #1, 1 = Binary rotary #0).

The binary rotary and variable background data buffers are placed at the end of the DPR area allowing the Data Gathering feature to start at the beginning of unreserved memory at 0xD200 (PMAC address). This PMAC (hard coded) memory management scheme is displayed below:



The varStart location is set by the variable of the same name in the registry. NC for Windows uses this for it's own internal use, setting aside n bytes of memory. You may consider using this memory space, from varStart to 0xDFFF for your own "internal" use as well. The space from varStart to the end of DPR, 0xDFFF, will be untouched by the Win32 Driver.

Using the DPR Control Panel

The PMAC provides an area in the DPR that emulates the PMAC's hardware control panel connector.

The PmacDPRControlPanel() function enables/disables the dual ported RAM control panel feature. When enabled, functions are available to:

- Jog+, Jog-, Pre-Jog , Home (Motors only).
- Run, Stop, Step and Feed Hold (Coordinate System only)
- Assign a feedrate override value for a particular coordinate system.

Four necessary conditions for any of the actions above to be realized are:

1. The DPR Control panel has been enabled with a call to PmacDPRControlPanel().
2. The request bit (or flag) for the specified action must first be set to 1 using the corresponding: PmacDPRSet*Action*Bit() function, where *Action* is any of those listed in I and II.
3. Any request bits that conflict with the one attempting to be realized must be reset to 0. For example if the "stop (or abort) bit" has been set and the programmer attempts to run a PMAC program by setting the "run bit", PMAC will not run the program due to the conflict in the requests.
4. The request_bit of the corresponding motor or coordinate system must be set to 1 by use of the PmacDPRSetRequestBit() function.

Action Bits

PMAC only reads action bits after the DPRSetRequestBit() function is called

None of the action request bits are written to by PMAC. Keeping track of which are set may be facilitated with the PmacDPRGet*Action*Bit() functions.

Request Bits

On the other hand, the request_bit is overwritten by PMAC. PMAC sets this bit to zero when it's finished acknowledging all the requested actions. Because of this fact, the programmer must be sure to wait long enough for PMAC to process the command before resetting the requested action bit to FALSE (in fact if your program is too fast PMAC may never see the request at all).

The request bit can therefore be considered a handshaking bit between the host computer and the PMAC. Host sets, PMAC notes the set bit, PMAC performs requested actions, PMAC resets the request bit.

Build routines that:

1. Set an action bit(s), via PmacDPRSet*Action*Bit()
2. Call the PmacDPRSetRequestBit() function specifying the appropriate Coordinate System or motor for which you want the requested action to occur.
3. Wait until the request bit is reset, via PmacDPRGetRequestBit()

4. Reset the appropriate action bit(s). This makes keeping track of request bits from within your application unnecessary.

An example implementation using "C" is shown below. Here the correct sequence of function calls is shown for jogging motor #1.

```
/* Turn on DPR Control Panel */
     PmacDPRControlPanel(device, TRUE);
/* Set the Jog positive request bit for specified motor to TRUE */
     PmacDPRSetJogPosBit(device,motor_num, 1);
.
  Other action requests
.
.
/* Set the enable-request bit for the specified motor to TRUE */
     PmacDPRSetRequestBit(device,motor_num,TRUE);
/* Wait until PMAC processes request */
     while(PmacDPRGetRequestBit(device,motor) && i<timeout)
         i++;
/* Reset jog bit to FALSE for housekeeping*/
     PmacDPsetJogPosBit(device,motor,FALSE);
```

Assigning A New Feedrate Override

The necessary conditions for assigning a coordinate system a new feedrate override (also known as "time-base") value are:

- Ix93, the Time Base Source Address for corresponding coordinate system must be set to it's default value.
- The DPR Control panel has been enabled with a call to PmacDPRControlPanel().
- The PmacDPRSetFOValue() must be called to set the feedrate override value for the specified coordinate system.
- The PmacDPRSetFOEnableBit() must be called to set the feedrate override enable bit for the same coordinate system specified in 2.

The *value* parameter for PmacDPRSetFOValue() may be anything from 1 to 32,767 in units of 1/32,768 msec. The *value* represents the "elapsed time" PMAC uses in its trajectory update equations each servo cycle. If it matches the true time, the trajectories will go at the programmed speeds. If it is greater than the true time, the trajectories will go faster, if it is less, they will go slower. *value* corresponds to values of 0 to 8,388,352 in units of I10 (1/8,388,608 msec). At the default I10 value of 3,713,707, this corresponds to a feedrate override (%) values from 0 to 225.87; for real-time execution (%100) *value* should be 14507.

Feedrate Override Enable Bits

As with the action request bits the feedrate override enable bits are also not written to by PMAC. PMAC continually writes the feedrate override value specified by PmacDPsetFOValue() when the corresponding enable bit is true. Keeping track of the feedrate override enable bit may be done via the PmacDPRGetFOEnableBit() function whose return value is the bit value.

Using the DPR Real Time Fixed Data Buffer

Startup/Shutdown and Handshaking Functions

PmacDPRRealTime()
PmacDPRGetHostBusyBit()
PmacDPRSetHostBusyBit()
PmacDPRGetPmacBusyBit()
PmacDPRGetServoTimer()

Data Query Functions

Global

```
PmacDPRGetServoTimer()
PmacDPRSysServoError()
PmacDPRSysReEntryError()
PmacDPRSysMemChecksumError()
PmacDPRSysPromChecksumError()
GLOBALSTATUS PmacDPRGetGlobalStatus()
```

Coordinate System

```
PmacDPRMotionBufOpen()
PmacDPRRotBufOpen()
PmacDPRGetFeedRateMode()
```

Motor

```
PmacDPRMotorServoStatus()
PmacDPRDataBlock()
PmacDPRPhasedMotor()
PmacDPRMotorEnabled()
BOOL PmacDPRHandwheelEnabled()
BOOL PmacDPROpenLoop()
BOOL PmacDPROnNegativeLimit()
BOOL PmacDPROnPositiveLimit()
void  PmacDPRSetJogReturn()
MOTION PmacDPRGetMotorMotion()

double PmacDPRGetCommandedPos()
double PmacDPRPosition()
double PmacDPRFollowError()
double PmacDPRGetVel()
void PmacDPRGetMasterPos()
void PmacDPRGetCompensationPos()

DWORD PmacDPRGetPrevDAC()
DWORD PmacDPRGetMoveTime()
```

The PmacDPRRealTime() function enables/disables the DPR real time fixed data buffer. When enabled all the other functions above may be used.

The **period** parameter in PmacDPRRealTime() specifies how often in servo-cycles PMAC will update the data in this buffer. The data for motor numbers 1-**n** will be updated in DPR where **n** is the parameter in PmacDPRSetMotors() function. The code segment below shows the sequence of calls (An excerpt from the provided PMACTEST application) that should be used to assure that reading of the DPR would not occur while PMAC is writing:

Notice the sequence for handshaking:

1. Inform PMAC that host is about to read the DPR by setting the Host Busy Bit via PmacDPRSetHostBusyBit()
2. See if PMAC is currently writing to the DPR via the DPRGetPMACBusyBit() routine, if so exit and try again next time
3. Call the Data Query functions to read the DPR Real Time Data buffer.
4. Inform PMAC that host is done reading and PMAC may update the Ram again by resetting the Host Busy bit

```
void PmacDPRRealTime::OnTimer(UINT nIDEvent)
{
        char buf[256];
// Call PmacDPRSetHostBusyBit
        m_pDoc->DPRSetHostBusyBit(TRUE);
        if(m_pDoc->DPRGetPMACBusyBit() )
             return;// Unlikely to get here
                                // But if it does, let's try
                                // again next time
        int servotimer = m_pDoc->DPRGetServoTimer();
        sprintf(buf,"%d",servotimer);
        m_ServoCounter.SetWindowText(buf);

        double aDouble = m_pDoc->DPRPosition(iCurrentMotor-1,1);
        sprintf(buf,"%11.1lf",aDouble);
        m_ActualPosition.SetWindowText(buf);

        aDouble = m_pDoc->DPRGetCommandedPos(iCurrentMotor-1,1);
        sprintf(buf,"%11.1lf",aDouble);
        m_CommandedPosition.SetWindowText(buf);

        aDouble = m_pDoc->DPRFollowError(iCurrentMotor-1,1);
        sprintf(buf,"%11.1lf",aDouble);
        m_FollowingError.SetWindowText(buf);

        m_pDoc->DPRGetCompensationPos(iCurrentMotor-1,1,&aDouble);
        sprintf(buf,"%11.1lf",aDouble);
        m_CompensationPosition.SetWindowText(buf);

        aDouble = m_pDoc->DPRGetVel(iCurrentMotor-1,1);
        sprintf(buf,"%11.1lf",aDouble);
        m_Velocity.SetWindowText(buf);

        long aLong = m_pDoc->DPRGetPrevDAC(iCurrentMotor-1);
        sprintf(buf,"%ld",aLong);
        m_PrevDAC.SetWindowText(buf);

        aLong = m_pDoc->DPRGetMoveTime(iCurrentMotor-1);
        sprintf(buf,"%ld",aLong);
        m_MoveTime.SetWindowText(buf);

        m_pDoc->DPRSetHostBusyBit(FALSE);
}
```

Using the DPR Background Fixed Data Buffer

Startup/Shutdown and Handshaking Functions

```
BOOL PmacDPRBackground(DWORD dwDevice, int on_off);
BOOL PmacDPRSetBackground(DWORD dwDevice);
void PmacDPRSetMotors(DWORD dwDevice,UINT n);
```

Data Query Functions

Motor Specific

```
double PmacDPRCommanded(DWORD dwDevice,int crd,char
                                                    axchar);
double PmacDPRVelocity(DWORD dwDevice,int motor,double
                                                    units);
double PmacDPRVectorVelocity(DWORD dwDevice,int num,int
```

```
                                                    motor[], double
                                                    units[]);
BOOL PmacDPRAmpEnabled(DWORD dwDevice,int motor);
BOOL PmacDPRWarnFError(DWORD dwDevice,int motor);
BOOL PmacDPRFatalFError(DWORD dwDevice,int motor);
BOOL PmacDPRAmpFault(DWORD dwDevice,int motor);
BOOL PmacDPROnPositionLimit(DWORD dwDevice,int motor);
BOOL PmacDPRHomeComplete(DWORD dwDevice,int motor);
BOOL PmacDPRInposition(DWORD dwDevice,int motor);
double PmacDPRGetTargetPos(DWORD dwDevice,int motor,
                                                    double posscale);
double  PmacDPRGetBiasPos(DWORD dwDevice,int motor,
                                                    double posscale);
```

Coordinate System Specific

```
long PmacDPRPe(DWORD dwDevice,int cs);
BOOL PmacDPRRotBufFull(DWORD dwDevice,int crd);
BOOL PmacDPRSysInposition(DWORD dwDevice,int crd);
BOOL PmacDPRSysWarnFError(DWORD dwDevice,int crd);
BOOL PmacDPRSysFatalFError(DWORD dwDevice,int crd);
BOOL PmacDPRSysRunTimeError(DWORD dwDevice,int crd);
BOOL PmacDPRSysCircleRadError(DWORD dwDevice,int crd);
BOOL PmacDPRSysAmpFaultError(DWORD dwDevice,int crd);
BOOL PmacDPRProgRunning(DWORD dwDevice,int crd);
BOOL PmacDPRProgStepping(DWORD dwDevice,int crd);
BOOL PmacDPRProgContMotion(DWORD dwDevice,int crd);
BOOL PmacDPRProgContRequest(DWORD dwDevice,int crd);
int   PmacDPRProgRemaining(DWORD dwDevice,int crd);
long  PmacDPRTimeRemInMove(DWORD dwDevice,int cs);
long  PmacDPRTimeRemInTATS(DWORD dwDevice,int cs);
```

Logical Query Functions

```
MOTIONMODE  PmacDPRGetMotionMode(DWORD dwDevice,int cs);
PROGRAM  PmacDPRGetProgramMode(DWORD dwDevice,int cs);
enum MOTION  PmacDPRGetMotorMotion(DWORD dwDevice,int motor);
```

To enable this feature call the PmacDPRBackground() routine with the on_off parameter set to a non-zero value. To retrieve the information use the provided DPR Background Fixed Data buffer query functions. Information for motors/coordinate systems 1-*n* will be updated where *n* is the second parameter of the PmacDPRSetMotors() function. You may call DPRSetMotors() before or after PmacDPRBackground().

When enabled all the query functions above may be used to:

1. Query PMAC's control panel, thumbwheel and machine I/O connector status.
2. Get motor target, bias and commanded position registers. Also query a motor's status word.
3. Query a coordinate systems status word, program status, program remaining, time remaining in move and acceleration and finally the coordinate systems currently executing line.

Once enabled call the DPRSetBackground() function to assure that the data query functions get fresh data. All data query functions have this function within it relieving the end user, you, from dealing with the handshaking details. A typical sequence of function calls to data would be:

```
PmacDPRSetMotors(p,4);// Update data for motors/cs 1 - 4
p  =  PMAC_DEVICE_NUMBER;
while(!DONE){
       printf("Target Position: %11.1lf\n",PmacDPRGetTargetPos(p, mtrcrd,1.0/(96.0*32.0)));
       printf("Bias Position: %11.1lf\n",PmacDPRGetBiasPos(p,mtrcrd,1.0/(96.0*32.0)));
       aDouble=PmacDPRCommanded(p,mtrcrd,'A');
```

```
        printf("Commanded Position A: %11.4lf\n",aDouble);
        aDouble=PmacDPRCommanded(p,mtrcrd,'B');
        printf("PmacCommanded Position B: %11.4lf\n",aDouble);
        aDouble=PmacDPRCommanded(p,mtrcrd,'C');
        printf("Commanded Position C: %11.4lf\n",aDouble);
        .
        .
        printf("WFE:%d\n",PmacDPRWarnFError(p,mtrcrd));
        printf("FFE:%d\n",PmacDPRFatalFError(p,mtrcrd));
        printf("AMP FAULT:%d\n",PmacDPRAmpFault(p,mtrcrd));
        printf("POSITION LIMIT:%d\n",PmacDPROnPositionLimit(p,mtrcrd));
        printf("HOME COMPLETE:%d\n",PmacDPRHomeComplete(p,mtrcrd));
        printf("MOTOR IN POSITION:%d\n",PmacDPRInposition(p,mtrcrd));
}
```

Using the DPR Variable Background Read/Write Data Buffer

Variable Read Data Buffer

The variable background data buffer can be used to have PMAC copy any of it's own internal registers into the DPR area. The registers copied are specified by using the startup functions discussed below. Up to 128 registers can be copied.

These routines may work in a multitasking environment, such as Windows. That is, multiple applications may use the Variable Background Data Buffer (VBGDB) simultaneously (for Prom versions 1.15G and above only, earlier proms are single user only), although the combined number of registers copied still remains at 128.

Startup/Shutdown

```
long PmacDPRVarBufInit(DWORD dwDevice, long num_entries, long
                                                    *addrarray);
```

PmacDPRVarBufInit() is used to initialize or append the VBGDB. If all goes well a non-zero "handle" is returned to the caller. The handle returned is necessary for all data querying functions and should therefore be stored for safekeeping. A return value of FALSE indicates initialization failed.

SeeMemory Organization/Correct Order of Initialization for a discussion on how memory is managed in the DPR and also the order in which buffers should be initialized and removed. This buffer should be added after all binary rotary buffers have been initialized, and removed before the binary rotary buffers are deleted if any exist.

The user must specify the number of entries to be updated by PMAC (**num_entries**) and all PMAC addresses to be copied into the DPR (the **addrarray** parameter). PMAC addresses are specified using an array of long integers. The most significant word of each long (upper 16 bits) specifies the word type. A value of 0, 1, 2 and 4 corresponds to Y, Long, X, and SPECIAL respectively. For Y, Long and X entries the least significant word specifies the actual PMAC address to be copied. For SPECIAL (PROM V 1.16 and beyond) the user must modify bits B13 and B15 for the type (0 = PLCC Ladder Diagram Function Blocks, 1..7 are for future expansion). When using the PLCC Ladder Diagram Function selection the least significant word specifies the function block structure member.

Take for example the following define statements and the initialization of a long integer array.

```
#define MTR1_ACTULPOS 0x00010028 /* PMAC Address D:$0028 */
#define MTR1_ACTULVEL 0x00020033 /* PMAC Address X:$0033 */
#define MTR1_DACOUT        0x00020045 /* PMAC Address X:$0045 */
#define FB_10              0x0004000A /* PMAC function block #10 */
long myaddrarr[3]={MTR1_DACOUT,MTR1_ACTULPOS,MTR1_ACTULVEL};
```

If myaddrarr is used as the last parameter in the PmacDPRVarBufInit() function the second parameter, **num_entries**, which is the number of elements in the long array would have to be 3.

Because only 128 entries may be updated, you may want to call PmacDPRGetVBGTotalEntries() to determine how many entries are already in use before calling PmacDPRVarBufInit(). Although PmacDPRVarBufInit() will also check to the available space and return an error if the request is too large.

The PmacDPRBackground() function enables/disables both background read data buffers (fixed and variable, a return value 1 means successful, 0 failure).

Once initialized a different set of PMAC registers to be copied to DPR may be specified by using the PmacDPRVarBufChange() function. This function works exactly as PmacDPRVarBufInit() except that it has one additional parameter, the handle to the already existing background data variable buffer

Shutting down the VBGDB is done by calling the PmacDPRVarBufRemove() function. This function deallocates memory within PComm32 and should always be called before any application using the VBGDB terminates.

Data Query Function Description

Once the variable buffer is initialized and enabled, data may be retrieved by use of the PmacDPRVarBufRead() function. The entry number, **entry_num**, specifies to PComm which of the elements in the address array passed to it, the calling function desires. In the example above, for instance, if we desired to retrieve motor 1 actual position, the second entry in the address array, the programmer would set entry_num to 1 (i.e. the first entry would be 0). The parameter **long_2** is a long pointer which should point to an memory element at least two long integers in size. One could use the following declared element for the **long_2** parameter:

```
long long_array[2];
```

If the data was ready when this function was called a value of TRUE is returned, and the data is placed in the **long_2** parameter. If the requested entry is a PMAC short word then only the first element in the long array is filled (i.e. long_array[0]) and the second element is not relevant. On the other hand if a PMAC long word or SPECIAL (PLCC Function block) is requested both entries in the long array are filled will data.

Note:

A PMAC Short word is, for example, an X or Y location (24 bits copied into 32 DPR bits). A Long word would be an L or D word (48 bits copied into 64 DPR bits).

PMAC long words (48 bits) will occupy two long words in DPR (64 bits, 2 "entries"). PmacDPRLFixed() or PmacDPRFloat() can be used to convert the long array to a meaningful numeric value. For example if 3 entries are to be updated and they are in this order:

1. Motor 1's DAC output (X:$0045)
2. Motor 1's actual position (D:$0028)
3. Motor 1's Actual velocity (X:$0033)

then entry_num=0 corresponds to motor 1's DAC output, entry_num = 1 correspond to the actual position registers least and most significant words and finally entry_num = 2 corresponds to motor 1's actual velocity register. The following demonstration source illustrates this process.

```
/* Get the dual word data */
      long data[2];
While(1){
      /* Get second entry*/
      if(PmacDPRVarBufRead(p,myVBGBHandle,1,data)){
            /* Convert to encoder counts (Assuming I108 = 96)*/
```

```
            MotorPosition = PmacDPRLFixed(data,1.0/(96.0*32.0));
            cprintf("\n\n Actual motor position after conversion
            =%lf\r\n",MotorPosition);
      }
}
```

All the handshaking necessary for this buffer is contained within DPRVarBufRead(). Simply poll the return value of this function to determine whether or not the data was ready.

Note:

If you are not using any of the Binary Rotary Buffers set dprVarStart, the initialization file parameter, to at most 0xDFFE. This way the last entry in your variable background data buffer will get updated.

PmacDPRGetVBGStartAddr()

Use this to attain the PMAC address of the start of the variable data buffer. This address also indicates the end of the free user memory.

PmacDPRBufLast()

This function determines which buffer was the last buffer entered in DPR.

Return Value:

0 = No Buffers in DPR
1 = Binary Rotary Buffer #1
2 = Binary Rotary Buffer #2
3 = Background Variable Buffer

Use this function to decide which buffer (either a binary rotary or the variable background data buffer) can be initialized or removed. The sequence for the binary rotary and variable background data buffers is like a last in first out stack. The order of initialization must be: 1. Binary rotary buffer 0, 2. Binary rotary buffer 1, 3. Variable background data buffer.

PmacDPRGetVBGServoTimer()

Returns the value of PMAC's servo timer

DPRGetVBGNumEntries()

Returns the number of items being updated in the buffer by the calling application. This number is the same as that passed to the function PmacDPRVarBufInit().

PmacDPRGetVBGTotalEntries()

Returns the number of items being updated in the buffer by all applications. Should never exceed 128.

DPRGetVBGAddress()

This function returns an entry within the address table initialized by PmacDPRVarBufInit(). For the first entry set entry_num to 0, the second set entry_num to 1 etc. PMAC reads this address table to determine what to copy into DPR memory. Typically not used by programmers.

DPRGetVBGDataOffset()

Returns the offset, in bytes, from the beginning of DPR that the data for this application's VBGDB begins. Typically not used by programmers.

DPRGetVBGAddrOffset()

Returns the offset, in bytes, from the beginning of DPR that the address array for this application's VBGDB begins. Divide by 4 to get the PMAC offset (from beginning of DPR 0xD000). Typically not used by programmers.

Variable Write Data Buffer

```
int PmacDPRWriteBuffer(DWORD dwDevice, int num_entries, struct
VBGWFormat *the_data);
```

This feature is available for PROM Version 1.15G and above.

The variable background write buffer can be used to have PMAC transfer up to 32 PMAC long or short words from DPR to its own internal memory without using the DPR ASCII feature. The data to be written into PMAC's memory is first placed in an array of VBGWFormat structures (definition shown below).

```
struct VBGWFormat{
      long type_addr;
      long data1;
      long data2;
};
```

The upper 16 bits of type_addr element specifies the type of data and the lower 16 bits specify the address to which data1 and data2 should be written.

The types of data that may be specified are as follows.

Bits 0-2 0/1/2/4 for Y/L/X/SPECIAL memory respectively. For SPECIAL (PROM V 1.16 and beyond) the user must modify bits B13 and B15 for the type (0 = PLCC Ladder Diagram Function Blocks, 1..7 are for future expansion).
Bits 3-7 Width = 1,4,8,12,16,20 (a value of 0 is 24 bit variable).
Bits 8-12 Offset = 0..23.
Bits 13-15 Special type.

The way in which one uses data1, and data2 is not intuitive. For Y or X memory data1 element will be used for specifying the 24 bits of data. The most significant byte of data1 and all of data 2 are irrelevant in this case. For L data, PMAC 48 bit word, data1 should hold the first 32 bits and data2 should hold the most significant 16 bits leaving the most significant 16 bits of data2 irrelevant.

TWS is not yet implemented.

PMAC addresses are specified using an array of long integers. The most significant word of each long (upper 16 bits) specifies the word type. A value of 0, 1, 2 and 4 corresponds to Y, Long, X, and SPECIAL respectively. For Y, Long and X entries the least significant word specifies the actual PMAC address to be copied. For SPECIAL (PROM V 1.16 and beyond) the user must modify bits B13 and B15 for the type (0 = PLCC Ladder Diagram Function Blocks, 1..7 are for future expansion). When using the PLCC Ladder Diagram Function selection the least significant word specifies the function block structure member.

Using the DPR Binary Rotary Motion Program Buffer

Startup/Shutdown Functions

```
PmacDPRRotBufInit()
PmacDPRRotBufRemove()
PmacDPRRotBufChange()
PmacDPRRotBufClr()
PmacDPRRotBuf()
PmacDPRBufLast()
```

Conversion & Transfer Functions

```
PmacDPRAsciiFileToRot()
PmacDPRBinaryFileToRot()
PmacDPRAsciiStrToRot()
PmacDPRBinaryToRot()
PmacDPRAsciiToBinaryFile()
PmacDPRAsciiStrToBinaryFile()
```

PmacDPRRotBufInit()

The PmacDPRRotBufInit() function is used to initialize the DPR Binary Rotary motion program buffers. Presently, there may be up to eight rotary buffers, buffer #0 through buffer #7. Buffers #0, and #1 correspond to coordinate system 1 and 2 respectively.

The size of a particular buffer may also be changed after PMAC driver initialization takes place by calling the () function. Note that using PmacDPRRotBufChange() will not cause the change in size to be persistent when resetting the operating system (registry is not changed). Note you should set I57 to 0 before changing the buffer size, and the buffer being changed should be the one last initialized.

The size of the rotary buffer in PMAC is typically half that allocated in the DP Ram. Two long words in the DP Ram, again, is mapped into 1 long PMAC word (48 bit).

The value assigned to DualPortRamRotx (where *x* = 1,2), which is in units of 32 bits, specifies how many instructions will fit into the buffer. Each ASCII instruction is ultimately transformed into a 64-bit code (from which PMAC will transform into a 48 bit operation code). A single instruction consists of a program command and it's argument. For example "X100" is a single instruction and "X1000Y1000Z1000" is 3 instructions. A simple formula to determine the bufsize as a function of the maximum number of buffered program statements is:

buffer_size = (number_of_motors_in_cs) * (num_lines) * (2);

The Rotary buffer size must be at least 6 words long. If too large then it may run into the gather buffer

Call PmacDPRRotBuf() to enable the Binary Rotary Buffer after it's been initialized.

PmacDPRRotBufRemove()

Use this function to remove any previously initialized rotary buffer.

PmacDPRRotBufChange()

Use this function to change the size of a rotary buffer. This will remove the existing rotary buffer, and allocate a new one in DPR. The binary rotary buffer you wish to change must be the last variable sized buffer to be created. If size is set to zero, this function is equivalent to PmacDPRRotBufRemove().

PmacDPRRotBufClr()

This function will clear either Binary Rotary buffers 0 or 1 in DPR (i.e. remove all entries). You should call disable the Binary Rotary Buffer first by calling PmacDPRRotBuf().

PmacDPRRotBuf()

Once initialized with PmacDPRRotBufInit() the DPRotBuf() function can be used to enable or disable the rotary buffer (if **onoff** = 1 then enable if 0 then disable). The return value is a boolean, TRUE for DPR feature enabled enabled, FALSE for disabled.

PmacDPRBufLast()

This function determines which buffer is the last variable sized buffer in DPR.

Return Value:

From 1.16B and beyond

9 = Background Variable Buffer
8 = Binary Rotary Buffer #7
7 = Binary Rotary Buffer #6
6 = Binary Rotary Buffer #5
5 = Binary Rotary Buffer #2
4 = Binary Rotary Buffer #4
3 = Binary Rotary Buffer #3
2 = Binary Rotary Buffer #2
1 = Binary Rotary Buffer #1
0 = No Buffers in DPR

Before 1.16B

3 = Background Variable Buffer
2 = Binary Rotary Buffer #2
1 = Binary Rotary Buffer #1
0 = No Buffers in DPR

Use this function to decide which buffer (either a binary rotary or the variable background data buffer) can be initialized or removed. The sequence for the binary rotary and variable background data buffers is like a last in first out stack. The order of initialization must be: 1. Binary rotary buffer 0, 2. Binary rotary buffer 1, 3. Variable background data buffer.

Downloading "Stuffing" Functions

There are several approaches to the conversion and downloading of PMAC ASCII program code:

1. PMAC ASCII program command strings either a) Created on the fly or b) Retrieved from a file are converted to binary and sent to the DPR rotary buffer via the PmacDPRAsciiStrToRot() function.
2. A PMAC ASCII file or series of strings is first converted to a binary file using PmacDPRAsciiToBinaryFile() or PmacDPRAsciiStrToBinaryFile() respectively. Then the binary file is then downloaded one line at a time using the DPRBinFileToRot() function.
3. PMAC ASCII program command strings either a) created on the fly or b) retrieved from a file are then converted to binary by use of the DPRStrcnvt() function. The binary result is placed into a union:

```
union {
  WORD   word_data[2*DPRCMDMAX+4];
  DWORD  long_data[DPRCMDMAX+2];
} pmac_cmd;
```

and is later placed in DPR via the DProtput() routine.

The Microsoft Visual C++ demo program, TESTAPP, illustrates the first of the three methods shown above.

In order for a rotary program to run several things must be done:

1. Configure a coordinate system in PMAC.
2. Close the servo loops too, and make sure all motors are ready to run a motion program (i.e. there not jogging homing etc.).
3. Initialize a rotary program buffer for the corresponding coordinate system.
4. Open PMAC's rotary buffer.
5. Begin execution of the rotary buffer (i.e. "B0R")

Items 1 and 2 may only have to be done once for a given application.

Return Codes Possible

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons, when this occurs the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

If you get something other than a DprBufBsy, DprOk, or DprEOF, I'd flag the user of the error. In this case the error is a conversion issue (converting to ASCII to BINARY).

PROGRAMMERS REFERENCE

Programmers Reference (Introduction)

The Programmers Reference of PComm32 details all of the PMAC library functions in groups of similar functionality. The description of each function includes the syntax, arguments, and possible return values. The groups are ordered as follows:

1. Initialization, Shutdown and Configuration Functions
2. ASCII Communication Functions
3. Downloading Functions
4. DPR Control Panel Functions
5. DPR Real Time Fixed Data Buffer
6. DPR Variable Background Data Buffer Functions
7. DPR Binary Rotary Buffer Functions
8. DPR Numeric Read / Write Functions
9. Interrupt Functions
10. Variable Functions
11. Data Types and Structures

Important Information About Method of Communication Being Used by PComm32

There are 3 methods by which PComm32 may be used to communicate to PMAC, over the Bus, Dual Ported Ram, or the Serial Port. Immediately after initialization (after a call to OpenPmacDevice()) the method used depends on what is stored in the system registry (either BUS or SERIAL). To change the method used at startup call the PmacConfigure() function so that the change is saved in the registry. Alternatively, if your operating system has the ability to configure drivers (i.e. in the control panel or the multimedia palette), go there and select the PMAC driver for configuration.

Maximum Number of Response Buffer Characters

Many of the ASCII Communication functions use an argument by the name of **maxchar**. The **maxchar** specifies the maximum number of characters that will be placed within a response buffer (a character string pointer also an argument) and should therefore never be larger than the size of the response buffer. Should PMAC's response be larger than the value specified by maxchar the remainder of the response will remain in PMAC's queue.

UNICODE Versus ANSI Compiles

The PComm32 supports both UNICODE and ANSI compiles. For a small portion of PMAC.DLL there are two sets of nearly identical functions, one for UNICODE and one for ANSI. The manual refers to the identical functions by using a "*X*" suffix. For example both the PmacGetResponseW() and PmacGetResponseA() functions are referred to in the manual as PmacGetResponse*X*().

For those folks programming in C/C++ (or any other program that supports include files and a macro preprocessor) you may keep your PComm32 specific code independent of the standard being used by using the definitions below (See UTIL.H).

```
#ifdef UNICODE

#define PmacGetResponse PmacGetResponseW
#define PmacGetControlResponse PmacGetControlResponseW
#define PmacSendChar PmacSendCharW
#define PmacSendLine PmacSendLineW
#define PmacGetLine PmacGetLineW
#define PmacGetBuffer PmacGetBufferW
```

```
#define PmacSendCommand PmacSendCommandW
#define PmacMultiDownload PmacMultiDownloadW
#define PmacAddDownloadFile PmacAddDownloadFileW
#define PmacRemoveDownloadFile PmacRemoveDownloadFileW
#define PmacRenumberFiles PmacRenumberFilesW
#define PmacGetErrorStr PmacGetErrorStrW

#else

#define PmacGetResponse PmacGetResponseA
#define PmacGetControlResponse PmacGetControlResponseA
#define PmacSendChar PmacSendCharA
#define PmacSendLine PmacSendLineA
#define PmacGetLine PmacGetLineA
#define PmacGetBuffer PmacGetBufferA
#define PmacSendCommand PmacSendCommandA
#define PmacMultiDownload PmacMultiDownloadA
#define PmacAddDownloadFile PmacAddDownloadFileA
#define PmacRemoveDownloadFile PmacRemoveDownloadFileA
#define PmacRenumberFiles PmacRenumberFilesA
#define PmacGetErrorStr PmacGetErrorStrA
#endif // !UNICODE
```

INITIALIZATION, SHUTDOWN AND CONFIGURATION FUNCTIONS

OpenPmacDevice()

```
BOOL OpenPmacDevice(DWORD dwDevice);
```

This function opens a channel for your program to use the PMAC driver.

In order for this function to succeed the PMAC Win32 Driver must be previously installed in the operating system. PMAC(*dwDevice*) must be registered in the environment. Once registered the system registry will contain the location and configuration of the PMAC specified by *dwDevice*. OpenPmacDevice looks to the registry for this information. The registry values are located in HKLM/System/CurrentControlSet/Services/Pmac/Device(*dwDevice*).

Every *OpenPmacDevice()* should be paired with a call to *ClosePmacDevice()* to release the resources used by the driver.

Arguments

dwDevice Device number to open.

Return Value

TRUE if success.

See Also

()

ClosePmacDevice()

```
BOOL ClosePmacDevice(DWORD dwDevice);
```

This function closes the channel your program to the PMAC driver.

Arguments

dwDevice Device number to close.

Return Value

TRUE if success.

See Also

()

PmacAbortDPRTest()

```
void PmacAbortDPRTest(DWORD dwDevice);
```

Stops the dual ported RAM test function initiated by the PmacDPRTest() function.

Arguments

dwDevice Device number to stop test for.

Return Value

TRUE if success.

See Also

PmacDPRTest()

PmacConfigure()

```
BOOL    PmacConfigure(HWND hwnd,DWORD dwDevice);
```

This function causes the PComm32 Driver's configuration dialog to be presented to the user for driver configuration. If the device is open it will be closed by this function. This function can be used to modify the many attributes associated with talking to a particular PMAC. These attributes are stored in the systems registry so that they will persist when the system recycles power or reboots.

Arguments

hwnd Handle to parent window for device configuration dialog.

dwDevice Device number to configure.

Return Value

TRUE if success.

PmacGetDpramAvailable()

```
BOOL    PmacGetDpramAvailable(DWORD dwDevice);
```

This function determines if dual ported RAM is available for use by your application. If the PMAC controller has dual ported RAM available and it was successfully initialized by the driver this function will return TRUE.

Arguments

dwDevice Device number.

Return Value

TRUE if Dual Ported RAM is available.

PmacInBootStrapMode()

```
BOOL PmacInBootStrapMode(DWORD dwDevice);
```

This function returns TRUE if the PMAC controller is in a mode known as BOOTSTRAP. It is in this mode that PMAC's equipped with flash ram will be able to accept a new firmware file. This mode is usually caused by a jumper being placed on the controller (i.e. E3 on PMAC2, E51 on PMAC).

This function is used in Delta Tau's Executive program when it first starts up. If the board is in BOOTSTRAP mode it changes the file menu option to allow for firmware download.

Arguments

dwDevice Device number.

Return Value

TRUE if the PMAC controller is in BOOTSTRAP mode.

See Also

`PmacDownloadFirmwareFile()`

PmacDPRTest()

```
LONG PmacDPRTest(DWORD dwDevice, DPRTESTMSGPROC msgp, DPRTESTPROGRESS
   prgp);
```

This function provides a method for Dual Ported RAM testing. Both progress and message callback functions are provided (See *Data Types, Structures, Callbacks, and Constants* in this manual for info on callback prototypes and descriptions).

NOTE:

This function will also set PMAC parameters I2,I5,I47,I48 and I49 in order to prevent the dual ported RAM from being interfered with from PMAC programs or firmware during the test.

Arguments

dwDevice	Device number.
msgp	Pointer to a user supplied DPRTESTMSGPROC procedure used for message reporting.
Prgp	Pointer to a user supplied DPRTESTPROGRESS procedure used for progress reporting.

Return Value

The number of errors encountered.

See Also

`PmacAbortDPRTest(), DPRTESTMSGPROC , DPRTESTPROGRESS.`

PmacSERIsOpen()

```
BOOL  PmacSERIsOpen(DWORD dwDevice);
```

Determines if there is a serial port channel currently open to a PMAC controller.

Arguments

dwDevice	Device number.

Return Value

TRUE if there is a valid serial device handle open to a PMAC.

See Also

`PmacSERGetHandle(), PmacSEROnLine().`

PmacSERGetBaudrate()

```
DWORD PmacSERGetBaudrate(DWORD dwDevice);
```

Gets the current serial port baud rate of the device.

Arguments

dwDevice	Device number.

Return Value

Current baud rate.

See Also

PmacSERSetBaudrate().

PmacSERGetHandle()

```
HANDLE PmacSERGetHandle(DWORD dwDevice);
```

Gets the current serial handle for the device.

Arguments

dwDevice Device number.

Return Value

Handle to a serial port. If port handle is not available then returns NULL(0).

See Also

`PmacSERIsOpen()`.

PmacSERGetPort()

```
DWORD PmacSERGetPort(DWORD dwDevice);
```

Gets the current serial port number for the device.

Arguments

dwDevice Device number.

Return Value

Port number. 1 = COM1 etc.

See Also

`PmacSERSetPort()`.

PmacSEROnLine()

```
BOOL PmacSEROnLine(DWORD dwDevice);
```

This function tests to see if there is a PMAC controller on the current serial port.

Arguments

dwDevice Device number.

Return Value

TRUE if a PMAC is found.

PmacSERSetBaudrate()

```
BOOL PmacSERSetBaudrate(DWORD dwDevice,DWORD
   baudrate);
```

Sets the current serial port baudrate of the device if it is different then the current value. Changes are made in the registry so that on driver initialization the last set baudrate is used. Communication is attempted using the new baudrate.

Arguments

dwDevice Device number.
baudrate Baud rate.

Return Value

TRUE if successfully established communication using the baudrate specified.

See Also

`PmacSERGetBaudrate(), PmacSERSetComm()`

PmacSERSetComm()

```
BOOL  PmacSERSetComm(DWORD dwDevice,DWORD
   port,DWORD baudrate,BOOL odd_parity);
```

This function sets the port number, baud rate and parity of the serial port for the device. Changes are made in the registry so that on driver initialization the last set attributes are used. Communication is attempted using the new settings.

Arguments

dwDevice	Device number.
port	Port number. 1 = COM1 etc.
baudrate	Baud rate.
odd_parity	Odd parity if TRUE else no parity.

Return Value

TRUE if successfully established communication using the settings specified.

See Also

`PmacSERSetPort(), PmacSERSetBaudrate`

PmacSERSetPort()

```
BOOL  PmacSERSetPort(DWORD dwDevice,DWORD port);
```

Sets the current serial port number for the device. Changes are made in the registry so that on driver initialization the last set attributes are used. Communication is attempted using the new port.

Arguments

dwDevice	Device number.
port	Port number. 1 = COM1 etc.

Return Value

TRUE if success.

See Also

`PmacSERGetPort(),PmacSERSetComm()`

ASCII COMMUNICATION FUNCTIONS

PmacFlush()

```
void PmacFlush(DWORD dwDevice);
```

PmacFlush() will cause PMAC to stop sending any messages it had started to send, even multi-line messages. The function then proceeds to strip any excess data left in the PMAC's output buffer queue. PmacFlush() also causes PMAC to empty the command buffer queue from the host, so it will erase any partially sent commands.

This can be useful before sending a query command for which you are expecting an exact response format. In particular if you are not sure what PMAC has been doing before, because it makes sure nothing else comes through before the expected response.

Arguments

dwDevice Device number.

Return Value

Nothing.

PmacGetAsciiComm()

```
enum ASCIIMODE PmacGetAsciiComm(DWORD dwDevice);
```

Returns the current communications mode. (BUS, DPRAM, SERIAL). Nearly all functions within the PComm32 driver library use this modal value to select the communication method to use. Use () to set this value.

Arguments

dwDevice Device number.

Return Value

ASCIIMODE enumeration.
0 = BUS
1 = DPRAM
2 = SERIAL

PmacGetBuffer*X*()

```
int PmacBUSGetBufferW(DWORD dwDevice,PWCHAR
   response,UINT maxchar,PUINT nc);

int PmacBUSGetBufferA(DWORD dwDevice,PCHAR
   response,UINT maxchar,PUINT nc);
```

PmacGetBuffer*X*() queries PMAC for a full multi-line response. Program listings, or memory dumps are examples of multi-line responses. This function will wait for one of the following conditions to occur before returning:

1. a *timeout* period
2. PMAC has the full response (as determined by the ACKnowledge character (ASCII 6) received at the end of the response)
3. The maximum number of characters, maxchar, was received

If PmacGetBuffer*X*() does not succeed on its first attempt to retrieve the response it may be that the PMAC device has no response available or a Timeout has occurred waiting for PMAC to respond. You can use the function () to first determine if a response is available.

Arguments

dwDevice	Device number.
response	Pointer to string buffer. PMAC's response is placed there by function.
maxchar	Maximum characters to copy into response parameter
nc	Pointer to an unsigned integer, the number of characters received is placed in there

Return Value

A status value indicating will be returned **in a future release**.

PmacGetControlResponse*X*()

```
int PmacGetControlResponseW(DWORD dwDevice,PWCHAR
   response,UINT maxchar, WCHAR ctl_char)

int PmacGetControlResponseA(DWORD dwDevice,PCHAR
   response,UINT maxchar, CHAR ctl_char)
```

Sends a control character to PMAC and returns a response, similar to PmacGetResponse*X*().

Arguments

dwDevice	Device number.
response	Pointer to string buffer. PMAC's response is placed there by function.
maxchar	Maximum characters to copy into response parameter
ctl_char	Control character to send to PMAC.

Return Value

Number of characters returned in response parameter.

PmacGetLine*X*()

```
int PmacGetLineA(DWORD dwDevice, PCHAR response,
   UINT maxchar);

int PmacGetLineW(DWORD dwDevice, PWCHAR response,
   UINT maxchar);
```

PmacGetLine*X*() queries PMAC for a line response. This function will wait for one of the following conditions to occur before returning:

1. a *timeout* period
2. PMAC has a response (A Carriage Return (ASCII 13) or an ACKnowledge character (ASCII 6) was received from PMAC)
3. The maximum number of characters, maxchar, was received

If PmacGetLine*X*() does not succeed on its first attempt to retrieve the response it may be that the PMAC device has no response available or a Timeout has occurred waiting for PMAC to respond. You can use the function () to first determine if a response is available.

Arguments

dwDevice	Device number.
response	Pointer to character buffer to receive character string from PMAC.
maxchar	Maximum number of characters to be received from PMAC.

Return Value

Number of characters received (including handshake characters).

FALSE (0) One of the following occurred:

1. No characters in PMAC's buffer queue
2. An error occurred

3. Communication not established

Error Codes

This routine initially sets the internal error flag to FALSE (0). Therefore calling PmacGetError() after this function will inform of any errors that occurred.

PmacGetResponse*X*()

```
int PmacGetResponseA(DWORD dwDevice,PCHAR
   response,UINT maxchar,PCHAR command);

int PmacGetResponseW(DWORD dwDevice,PWCHAR
   response,UINT maxchar,PWCHAR command);
```

Most if not all of your communication with the PMAC can be handled with the PmacGetResponse*X*() function. This function will send a command string (i.e. "#1j+, "?", "Open Prog1", etc.) to the PMAC and retrieve and place any pending responses within a response buffer for your use. This is an efficient and *safe* function to use. The word *safe* is emphasized because there are functions (i.e. PmacSendChar*X*(), PmacGetLine*X*() and PmacSendLine*X*()) which, if not used carefully can cause un-synchronized communication (the mismatching of PMAC Commands to PMAC responses). PmacGetResponse*X*() always matches the command string with the response string or else it "times out".

Arguments

dwDevice	Device number.
response	Pointer to string buffer to copy the PMAC's response into.
maxchar	Maximum characters to copy.
command	Pointer to NULL terminated string to be sent to the PMAC as a question/command.

Return Value

If successful this function returns the number of characters received including handshake characters. Otherwise FALSE (0) which implies of course that an error occurred, or no characters were received since PMAC was not required to respond.

Error Codes

This routine initially sets the internal error flag to FALSE (0). Therefore calling PmacGetError() after this function will inform of any errors that occurred.

PmacReadReady()

```
BOOL PmacReadReady(DWORD dwDevice);
```

Determines if PMAC has a response waiting on the current communications channel (BUS, DPRAM, SERIAL). Use this function before using other query functions (PmacGetLine*X*() or PmacGetBuffer*X()* for example) to prevent Timeout errors thus speeding communication through put.

Arguments

dwDevice	Device number.

Return Value

TRUE if PMAC has a response waiting.

PmacSendChar*X*()

```
BOOL PmacSendCharA(DWORD dwDevice,CHAR outchar);
BOOL PmacSendCharW(DWORD dwDevice,WCHAR outchar);
```

Sends a single character to the PMAC. Does not append EOL (Carraige Return for PMAC).

Arguments

dwDevice	Device number.
outchar	Character sent to PMAC.

Return Value

TRUE if success.

PmacSendCommand*X*()

```
void PmacSendCommandA(DWORD dwDevice,PCHAR
   command);
void PmacSendCommandW(DWORD dwDevice,PWCHAR
   command);
```

Sends a string buffer to PMAC and flushes out any response from PMAC. Also if PMAC is responding to any previous communication exchange it is cleared.

> ***Note:***
>
> This function is robust but slow since it will timeout at least once to assure there is nothing left in PMAC's output ASCII queue. **It is better to use GetResponse() instead.**

Arguments

dwDevice	Device number.
command	Pointer to NULL terminated string sent to PMAC.

Return Value

Nothing.

PmacSendLine*X*()

```
int PmacSendLineA(DWORD dwDevice,PCHAR command);
int PmacSendLineW(DWORD dwDevice,PWCHAR command);
```

Sends a string buffer to PMAC and disregards any response from PMAC.

Arguments

dwDevice	Device number.
command	Pointer to NULL terminated string sent to PMAC.

Return Value

Number of characters sent.

PmacSetAsciiComm()

```
BOOL PmacSetAsciiComm(DWORD dwDevice,enum
   ASCIIMODE m);
```

Used to set the current communications mode if available.

Note:

This will not change the communication method used when PComm32 starts upon initialization. To do this see PmacConfigure().

Arguments

dwDevice	Device number.
m	Mode: 0 = BUS, 1 = DPRAM, 2 = SERIAL.

Return Value

TRUE if success.

See Also

PmacConfigure()

DOWNLOADING FUNCTIONS

PmacAbortDownload()

```
void PmacAbortDownload(DWORD dwDevice);
```

Calling this function will cause a download in progress to be aborted. This applies for driver downloading functions PmacDownload(), PmacDownloadFile() and PmacDownloadFirmwareFile().

Arguments

dwDevice Device number.

Return Value

Nothing.

See Also

PmacDownload(), PmacDownloadFile() and PmacDownloadFirmwareFile()

PmacAddDownloadFile*X*()

```
int PmacAddDownloadFileA(DWORD dwDevice,PCHAR
   inifile,PCHAR szUserId,PCHAR szDLFile);
int PmacAddDownloadFileW(DWORD dwDevice,PWCHAR
   inifile,PWCHAR szUserId,PWCHAR szDLFile);
```

This utility function was added to assist in the use of PmacMultiDownload*X*() function. It appends a specified file name to the file list within the initialization file, *inifile*. If the file to be added to the list already exists the initialization file will remain unchanged. To remove a file from the list use ().

All parameters to this function are case insensitive and white space characters will be ignored.

Arguments

inifile Initialization file to add new download file to. If NULL, the default is "WIN.INI".
szUserId User id string, this associates the file being added to a given application or user.
szDLFIle Name of the file to add to download list.

Return Value

Greater than zero(0) if successful, else zero(0);

See Also

PmacMultiDownload(), ()

PmacCompilePLCC*X*()

```
short PmacPlccCompilerA(DWORD dwDevice,UINT
   *datainfo,PCHAR inName,PCHAR outName);
short PmacPlccCompilerW(DWORD dwDevice,UINT
   *datainfo,PWCHAR inName,PWCHAR outName);
```

Converts a PMAC PLC program to a PMAC PLCC program.

Arguments

dwDevice Device number.
datainfo Pointer to store line number where an error was detected if return value was non-zero. Or the number of 56K memory locations used to produce the PLCC if the return value = zero(0)
inName Path of the input file.
outName Path of the file to create.

Return Value
Zero(0) - when PLC compiled okay. Greater than zero(0) - when PLC did not compile okay.

PmacDownload()

```
int  PmacDownloadA(DWORD dwDevice,
   DOWNLOADMSGPROC msgp, DOWNLOADGETPROC getp,
   DOWNLOADPROGRESS pprg, PCHAR filename, BOOL
   macro, BOOL map, BOOL log, BOOL dnld);

int  PmacDownloadW(DWORD dwDevice,
   DOWNLOADMSGPROC msgp, DOWNLOADGETPROC getp,
   DOWNLOADPROGRESS pprg, PWCHAR fname, BOOL
   macro, BOOL map, BOOL log, BOOL dnld);
```

Download function that sends a macro file to PMAC. If you want the simplest form of sending a file to the PMAC use the function ().

Arguments
dwDevice	Device number.
msgp	Pointer to message procedure pointer. (DOWNLOADMSGPROC) If NULL no function is called.
getp	Pointer to line retrieval function. (DOWNLOADGETPROC) If NULL no function is called.
pprg	Pointer to download progress function. (DOWNLOADPROGRESS) If NULL no function is called.
filename	Path of file to download.
macro	Flag to parse for macros.
map	Flag to create a map file created from macros.
log	Flag to create a log file. This is the same messages as sent to the “msgp” procedure.
dnld	Flag indicating to send final parsed file to the PMAC.

Return Value
The error count. Zero(0) if no errors.

See Also
DOWNLOADMSGPROC, DOWNLOADPROGRESS, (), "Downloading To PMAC" in Users Manual.

PmacDownloadFile()

```
void PmacDownloadFile(DWORD dwDevice,char
   *fname);
```

Sends a file to PMAC without parsing. Straight from disk to the PMAC. If you want macro parsing use the PmacDownload() function.

Arguments
dwDevice	Device number.
fname	Path of file to download.

Return Value
Nothing.

See Also
"Downloading To PMAC" in Users Manual.

PmacDownloadFirmwareFile()

```
int  PmacDownloadFirmwareFile(DWORD dwDevice,
   DOWNLOADMSGPROC msgp, DOWNLOADPROGRESS prgp,
   PCHAR filename);
```

Flash ROM installation function. PMAC must be in BOOTSTRAP mode in order to download new firmware. Usually a jumper (E3 on PMAC2, E51 on PMAC) puts PMAC in a BOOTSTRAP mode. If talking serially the baudrate must be set to 38400.

Arguments

dwDevice	Device number.
Msgp	Pointer to message procedure pointer. (DOWNLOADMSGPROC) If NULL no function is called.
prgp	Pointer to download progress function. (DOWNLOADPROGRESS) If NULL no function is called.
filename	Path of firmware binary file to download.

Return Value

The error count. Zero(0) if no errors.

PmacMultiDownload()

```
int PmacMultiDownload(DWORD dwDevice,
   DOWNLOADMSGPROC msgp,  PCHAR outfile, PCHAR
   inifile, PCHAR szUserId, BOOL macro, BOOL
   map, BOOL log, BOOL dnld);
```

This function will resolve the Multi-Application problem associated with PMAC compiled PLCs. Similar to the Download() routine, a file is downloaded to PMAC. The difference here however is that the names of new files are stored in the WIN.INI file.

For example the syntax within the WIN.INI file may look like this:

Note:

The use of this function is **required** in those systems in which more than one application will be downloading PLCCs to the PMAC. The issue here is that all PLCCs must be downloaded at the same time.

```
[PMACDOWNLOAD]
DOWNLOADFILE1="WINDEMO C:\PCOMM\PCOMM\TESTPLCC.PLC"
DOWNLOADFILE2="WINDEMO C:\PCOMM\PCOMM\NCPANEL.PLC"
DOWNLOADFILE3="WINDEMO C:\PCOMM\WIZDOM\SPINDLE.PLC"
DOWNLOADFILE4="PEWIN C:\PCOMM\WIZDOM\SPINDLE.PLC"
DOWNLOADFILE5="USERPROGRAM C:\USER3\BIFF.PLC"
```

This function has added flexibility with the use of the szUserId parameter (described in the table below). MultiDownload() makes a temporary file, by the name of **outfile**, and places an include statement for every file within the initialization file, **iniFile**. It then calls the download() function described above. The **outfile** parameter will also be the name of any intermediary files created during the download, such as the log, map, and, compiled files (i.e. files produced will be **outfile**.log **outfile**.map etc.).

Note:

All files that include other files need to use the fully qualified path. For example if one of the files that to be downloaded is myfile.plc, and within myfile.plc there is an include statement for myfile.h, the include statement must use the fully qualified path (i.e. #include "c:\mydirect\myfile.h")

Arguments

outfile	The file name for which the output files will be named. This parameter also specifies the path of the files generated by this function. If NULL, the default is "PMACDNLD", path of file is active directory. It's best to supply a path for a consistent output path. This also indirectly specifies the name of the log, map and other intermediary files that will be created.
IniFile	Initialization file to be used. If Null the default is "WIN.INI".
user	Application user id string. If this is NULL all within the iniFile are downloaded within "PMACDOWNLOAD" section. Otherwise, only those preceded by the "user id" string within the iniFile will be downloaded.
macro	Flag to parse for macros.
map	Flag to create a map file created from macros.
log	Flag to create a log file. This is the messages sent to “msgp”.
dnld	Flag `indicating to send final parsed file to PMAC.`

Return Value

TRUE, if all went down well, otherwise -1 * (Number of errors) or FALSE if memory or file could not be allocated/opened, or if WIN.INI did not contain files to be downloaded.

See Also

PmacAddDownloadFile*X*, PmacRemoveDownloadFile*X*, "Downloading To PMAC" in the Users Manual.

PmacRemoveDownloadFile*X*()

```
int PmacRemoveDownloadFile(DWORD dwDevice, PCHAR
   inifile, PCHAR szUserId, PCHAR szDLFile);
```

This utility function was added to assist in the use of the PmacMultiDownload*X*() function. It removes a specified file name from the file list within the initialization file (also specified). It also renumbers the remaining files into a continuous block. It also may remove all files associated with a particular id string, by setting the *szDLFile* parameter to NULL.

All parameters to this function are case insensitive and white space characters will be ignored.

To add a file from the list, use ().

Arguments

inifile	Initialization file to remove file from. If NULL, the default is "WIN.INI"
szUserId	User id string, this associates the file being added to a given application or user.
SzDLFIle	Name of the file to remove from download list. If Null, it will remove all files with the szUserId tag.

Return Value

TRUE if file located and removed, else FALSE.

See Also

PmacAddDownloadFile*X*(), PmacRemoveDownloadFile*X*(), "Downloading To PMAC" in the Users Manual.

PmacSetMaxDownloadErrors()

```
void PmacSetMaxDownloadErrors(UINT max);
```

Sets the limit of errors allowed during a download before the function will automatically abort. Valid for use with functions, PmacDownload(), PmacDownloadFile(), and PmacMultiDownload().

Arguments

max	Number of errors allowed before download will terminate

Return Value

Nothing.

See Also

PmacDownload(), PmacDownloadFile(), PmacMultiDownload(), and "Downloading To PMAC" in the Users Manual.

PmacThreadDownload*X*()

```
int PmacThreadDownloadA(DWORD dwDevice,
   DOWNLOADMSGPROC msgp, DOWNLOADGETPROC getp,
   DOWNLOADPROGRESS pprg, PCHAR filename, BOOL
   macro, BOOL map, BOOL log, BOOL dnld);

int PmacThreadDownloadW(DWORD dwDevice,
   DOWNLOADMSGPROC msgp, DOWNLOADGETPROC getp,
   DOWNLOADPROGRESS pprg, PWCHAR fname, BOOL
   macro, BOOL map, BOOL log, BOOL dnld);
```

Download function that sends a macro file to PMAC using its own thread.

The callback functions provided must be aware of the asynchronous nature of the callbacks.

If you want the simplest form of sending a file to the PMAC without macro parsing use the function PmacDownloadFile().

Arguments

dwDevice	Device number.
msgp	Pointer to message procedure pointer. (DOWNLOADMSGPROC) If NULL no function is called.
getp	Pointer to line retrieval function. (DOWNLOADGETPROC) If NULL no function is called.
pprg	Pointer to download progress function. (DOWNLOADPROGRESS) If NULL no function is called.
filename	Path of file to download.

macro	Flag to parse for macros.
map	Flag to create a map file created from macros.
log	Flag to create a log file. This is the same messages as sent to the “msgp” procedure.
dnld	Flag indicating to send final parsed file to the PMAC.

Return Value

Zero(0) if the thread did not start.

See Also

DOWNLOADMSGPROC, DOWNLOADPROGRESS, "Downloading To PMAC" in the Users Manual.

DUAL PORTED RAM CONTROL PANEL

PmacDPRControlPanel()

```
BOOL PmacDPRControlPanel(DWORD dwDevice, int
   onoff);
```

This function is used to enable or disable the PMAC DPR Control Panel feature.

Arguments

dwDevice	Device number.
onoff	To turn on use TRUE (1) else FALSE (0)

Return Value

This function returns the active status of the DPR Control Panel Feature. TRUE(1) is on, FALSE (0) is off.

See Also

"Using the DPR Control Panel" in the Users Manual, PmacDPRGet*Action*Bit() and PmacDPRSet*Action*Bit()

PmacDPRGet*Action*Bit() and PmacDPRSet*Action*Bit()

```
void PmacDPRSetJogPosBit(DWORD dwDevice, int
   motor, int onoff);
int PmacDPRGetJogPosBit(DWORD dwDevice, int
   motor);
void PmacDPRSetJogNegBit(DWORD dwDevice, int
   motor, int onoff);
int PmacDPRGetJogNegBit(DWORD dwDevice, int
   motor);
void PmacDPRSetJogReturnBit(DWORD dwDevice,int
   motor,int onoff);
int PmacDPRGetJogReturnBit(DWORD dwDevice, int
   motor);
void  PmacDPRSetRunBit(DWORD dwDevice, int cs,
   int onoff);
int PmacDPRGetRunBit(DWORD dwDevice, int cs);
void PmacDPRSetStopBit(DWORD dwDevice, int cs,
   int onoff);
int PmacDPRGetStopBit(DWORD dwDevice, int cs);
void PmacDPRSetHomeBit(DWORD dwDevice, int cs,
   int onoff);
int PmacDPRGetHomeBit(DWORD dwDevice, int cs);
void PmacDPRSetHoldBit(DWORD dwDevice, int cs,
   int onoff);
int PmacDPRGetHoldBit(DWORD dwDevice, int cs);
int PmacDPRGetStepBit(DWORD dwDevice, int cs);
void PmacDPRSetStepBit(DWORD dwDevice, int cs,
   int onoff);
```

These functions are used to control the PMAC DPR Control Panel feature. The PmacDPRSet*Action*Bit() and PmacDPRGet*Action*Bit() routines set/get a bit in DPR which indicate to PMAC to perform the an action on the designated motor/coordinate system.

Arguments

dwDevice	Device number.
onoff	To turn on use TRUE (1) else FALSE (0)

Return Value
The PmacDPRGet*Action*Bit() routines return the state of the bit (0/1). The PmacDPRSet*Action*Bit() functions return nothing.

See Also
"Using the DPR Control Panel" in the Users Manual, PmacDPRControlPanel()

PmacDPRGetFOEnableBit() and PmacDPRSetFOEnableBit()

```
int PmacDPRGetFOEnableBit(DWORD dwDevice, int
   cs);

void PmacDPRSetFOEnableBit(DWORD dwDevice, int
   cs, int on_off);
```

These functions are used to control the PMAC DPR Control Panel feature. The PmacDPRSetFOEnableBit() and PmacDPRGetFOEnableBit() routines set/get a bit in DPR which tell PMAC whether or not to use the feedrate specified in the DPR (see). PMAC will not reset the request bit. The value of the feedrate is specified using PmacDPRSetFOValue().

Arguments

dwDevice	Device number.
mtr_crd	Motor or Coordinate System Number (1-8)
onoff	To turn on use TRUE (1) else FALSE (0)

Return Value
The PmacDPRGetRequestBit() routines return the state of the bit (0/1). The PmacDPRSetRequestBit() functions return nothing.

See Also
"Using the DPR Control Panel" in the Users Manual, PmacDPRControlPanel(), PmacDPRSetFOValue()

PmacDPRGetFOValue() and PmacDPRSetFOValue()

```
void PmacDPRSetFOValue(DWORD dwDevice, int cs,
   int value);

int PmacDPRGetFOValue(DWORD dwDevice, int cs);
```

These functions are used to control the PMAC DPR Control Panel feature. The PmacDPRSetFOValue() and PmacDPRGetFOValue() routines set/get the feedrate override value in DPR. Whether or not the feedrate is used depends on the arguments used in the last call to PmacDPRSetFOEnableBit().

The *value* parameter for PmacDPRSetFOValue() may be anything from 1 to 32,767 in units of 1/32,768 msec. The *value* represents the "elapsed time" PMAC uses in its trajectory update equations each servo cycle. If it matches the true time, the trajectories will go at the programmed speeds. If it is greater than the true time, the trajectories will go faster, if it is less, they will go slower. *value* corresponds to values of 0 to 8,388,352 in units of I10 (1/8,388,608 msec). At the default I10 value of 3,713,707, this corresponds to a feedrate override (%) values from 0 to 225.87; for real-time execution (%100) *value* should be 14507.

Arguments

dwDevice	Device number.
cs	Coordinate System Number (1-8)
value	See explanation above

Return Value

The PmacDPRGetRequestBit() routines return the state of the bit (0/1). The PmacDPRSetRequestBit() functions return nothing.

See Also

"Using the DPR Control Panel" in the Users Manual, PmacDPRControlPanel(), PmacDPRSetFOValue()

PmacDPRGetRequestBit() and PmacDPRSetRequestBit()

```
int PmacDPRGetRequestBit(DWORD dwDevice, int
   mtr_crd);

void PmacDPRSetRequestBit(DWORD dwDevice,int
   mtr_crd,int onoff);
```

These functions are used to control the PMAC DPR Control Panel feature. The PmacDPRSetRequestBit() and PmacDPRGetRequestBit() routines set/get a bit in DPR which indicate to PMAC to look at all the *action* bits that have been set and act on them. PMAC will reset the request bit once the requested actions have been processed. The request bit can therefore be considered a handshaking bit between the host computer and the PMAC. Host sets, PMAC notes the set bit, PMAC performs requested actions, PMAC resets the request bit.

Arguments

dwDevice	Device number.
mtr_crd	Motor or Coordinate System Number (1-8)
onoff	To turn on use TRUE (1) else FALSE (0)

Return Value

The PmacDPRGetRequestBit() routines return the state of the bit (0/1). The PmacDPRSetRequestBit() functions return nothing.

See Also

"Using the DPR Control Panel" in the Users Manual, PmacDPRControlPanel()

DPR REAL TIME FIXED DATA BUFFER

DPR Real Time Fixed Data Buffer Initialization and Handshaking

Initialization

```
BOOL PmacDPRRealTime(DWORD dwDevice, UINT period,
   int on_off);
void PmacDPRSetMotors(DWORD dwDevice,UINT n);
```

HandShaking

```
void PmacDPRSetHostBusyBit(DWORD dwDevice,int
   on_off);
int  PmacDPRGetHostBusyBit(DWORD dwDevice);
int  PmacDPRGetPmacBusyBit(DWORD dwDevice);
```

PmacDPRRealTime() starts or stops the PMAC's automatic write into the DPR real time fixed variable buffer. PMAC will copy motor information for motors 1-n where n is the argument of the PmacDPRSetMotors() function. You may call DPRSetMotors() before or after PmacDPRRealTime().

The PmacDPRRealTime() function above enables/disables the DP Ram real time fixed data buffer. When enabled all the other query functions may be used.

The **period** parameter in PmacDPRRealTime() specifies how often in servo-periods PMAC will update the data in this buffer. The data for motor numbers 1-**n** will be updated in DPR where **n** is the parameter in PmacDPRSetMotors() function.

Note the correct sequence for handshaking:

1. Inform PMAC that host is about to read the DPR by setting the Host Busy Bit via PmacDPRSetHostBusyBit()
2. See if PMAC is currently writing to the DPR via the DPRGetPMACBusyBit() routine, if so exit and try again next time
3. Call the Data Query functions to read the DPR Real Time Data buffer.
4. Inform PMAC that host is done reading and PMAC may update the Ram again by resetting the Host Busy bit via PmacDPRSetHostBusyBit().

Arguments

dwDevice	Device number.
period	In units of servo periods (See I10 in PMAC Users Manual)
on_off	Turn on (1) or off (0)

DPR Real Time Fixed Data Buffer Query Routines

Global

```
int  PmacDPRGetServoTimer(DWORD dwDevice);
```

The return value of this routine reflects PMAC's servo counter register located at PMAC address 0.

```
BOOL PmacDPRSysServoError(DWORD dwDevice);
```

The return value of this function indicates if PMAC could not properly complete its servo routines. This is a serious error condition. It returns FALSE if the servo operations have been completing properly.

```
BOOL PmacDPRSysReEntryError(DWORD dwDevice);
```

Returns TRUE if a real-time interrupt task has taken long enough so that it was still executing when the next real-time interrupt came (I8+1 servo cycles later). It stays TRUE until the card is reset or the bit in the global status register is reset manually.

```
BOOL PmacDPRSysMemChecksumError(DWORD dwDevice);
```

The return value is TRUE if a checksum error has been detected for either the PMAC firmware or the user program buffer space. PmacDPRSysPromChecksumError() distinguishes between the two cases.

```
BOOL PmacDPRSysPromChecksumError(DWORD dwDevice);
```

Returns TRUE if a firmware checksum error has been detected in PMAC's memory. The return value is FALSE if a user program checksum has been detected, or if no memory checksum error has been detected. PmacDPRSysMemChecksumError() distinguishes between the two cases.

```
GLOBALSTATUS PmacDPRGetGlobalStatus(DWORD dwDevice);
```

Returns a global status structure, GLOBALSTATUS, defined in the section "Data Types, Structures, Callbacks, and Constants.

Coordinate System

```
BOOL PmacDPRMotionBufOpen(DWORD dwDevice);
```

The return value indicates whether or not a motion program buffer is currently open.

```
BOOL PmacDPRRotBufOpen(DWORD dwDevice);
```

The return value indicates whether or not a rotary buffer is currently open.

double PmacDPRGetFeedRateMode(DWORD dwDevice,int coordinate_system, BOOL *mode);

The return value of this function is TRUE if programmed moves in this coordinate system are currently specified by time (TM or TA), and the move speed is derived. It is 0 if programmed moves in this coordinate system are currently specified by feedrate (speed; F) and the move time is derived.

Motor

```
SERVOSTATUS PmacDPRMotorServoStatus(DWORD dwDevice,int motor);
```

Returns a global status structure, SERVOSTATUS, defined in the section "Data Types, Structures, Callbacks, and Constants".

```
BOOL CALLBACK PmacDPRDataBlock(DWORD dwDevice,int motor);
```

This function returns TRUE when move execution has been aborted because the data for the next move section was not ready in time. This is due to insufficient calculation time. It is FALSE otherwise.

```
BOOL PmacDPRPhasedMotor(DWORD dwDevice,int motor);
```

This function returns TRUE when Ix01 is 1 and this motor is being commutated by PMAC, it is FALSE when Ix01 is 0 and this motor is not being commutated by PMAC.

```
BOOL PmacDPRMotorEnabled(DWORD dwDevice,int motor);
```

This function returns TRUE when Ix00 is 1 and the motor calculations are active. It is 0 when Ix00 is 0 and motor calculations are deactivated.

```
BOOL PmacDPRHandwheelEnabled(DWORD dwDevice,int motor);
```

This function returns TRUE when Ix06 is 1 and position following for this axis is enabled. It returns FALSE when IX06 is 0 and position following is disabled.

```
BOOL PmacDPROpenLoop(DWORD dwDevice,int motor);
```

When this function returns TRUE the servo loop for the motor is open, either with outputs enabled or disabled (killed). It returns FALSE when the servo loop is closed (under position control, always with outputs enabled).

```
BOOL PmacDPROnNegativeLimit(DWORD dwDevice,int motor);
```

Returns TRUE when motor actual position is less than the software negative position limit (Ix14), or when the hardware limit on this end (+LIMn – note!) has tripped; it is FALSE otherwise.

```
BOOL PmacDPROnPositiveLimit(DWORD dwDevice,int motor);
```

Returns TRUE when motor actual position is greater than the software positive position limit (Ix13), or when the hardware limit on this end (-LIMn – note!) has tripped; it is FALSE otherwise.

```
void  PmacDPRSetJogReturn(DWORD dwDevice,int motor);
```

This can be used to set the Jog Return Position for the motor specified. The current actual position is used to assign the Jog Return Position.

```
MOTION PmacDPRGetMotorMotion(DWORD dwDevice,int motor);
```

Returns an enumeration based on the motion state of the specified motor.

```
typedef enum { inpos,jog,running,homing,handle,openloop,disabled }
MOTION;
```

```
double PmacDPRGetCommandedPos(DWORD dwDevice,int motor, double units);
```

This function returns the commanded position of the specified motor. Units are in encoder counts unless the parameter *units* in not one.

```
double PmacDPRPosition(DWORD dwDevice,int i,double units);
```

This function returns the actual position of the specified motor in units of encoder counts provided the *units* variable is unity.

```
double PmacDPRFollowError(DWORD dwDevice,int motor,double units);
```

This function returns the following error of the specified motor in units of encoder counts provided the *units* parameter is unity.

```
double PmacDPRGetVel(DWORD dwDevice,int motor,double units);
```

This function returns the velocity of the specified motor in units of (1/(Ix09*32) counts per servo cycle.

```
void PmacDPRGetMasterPos(DWORD dwDevice,int motor,double units,double
*the_double);
```

This function returns the master position of the specified motor in units of encoder counts unless the *units* parameter is unity.

```
void PmacDPRGetCompensationPos(DWORD dwDevice,int motor,double
units,double *the_double);
```

This function returns the compensation position of the specified motor in units of encoder counts unless the *units* parameter is unity.

```
DWORD PmacDPRGetPrevDAC(DWORD dwDevice,int motor);
```

This function returns the DAC value of the previous servo cycle. It is in units of 1/256 DAC bits.

```
DWORD PmacDPRGetMoveTime(DWORD dwDevice,int motor);
```

This function returns the time (in milliseconds) left in the currently executing move.

DPR BACKGROUND FIXED DATA BUFFER

DPR Background Fixed Data Buffer Initialization and Handshaking

```
BOOL PmacDPRBackground(DWORD dwDevice, int
   on_off);
BOOL PmacDPRSetBackground(DWORD dwDevice);
void PmacDPRSetMotors(DWORD dwDevice,UINT n);
```

To enable this feature call the PmacDPRBackground() routine with the on_off parameter set to a non-zero value. To retrieve the information use the provided DPR Background Fixed Data buffer query functions. Information for motors/coordinate systems 1-*n* will be updated where *n* is the second parameter of the PmacDPRSetMotors() function. You may call DPRSetMotors() before or after PmacDPRBackground().

When enabled all the query functions above may be used to:

1. Query PMAC's control panel, thumbwheel and machine I/O connector status.
2. Get motor target, bias and commanded position registers. Also query a motor's status word.
3. Query a coordinate systems status word, program status, program remaining, time remaining in move and acceleration and finally the coordinate systems currently executing line.

Once enabled call the DPRSetBackground() function to assure that the data query functions get fresh data. All data query functions have this function within it relieving the end user, you, from dealing with the handshaking details. A typical sequence of function calls to data would be:

```
PmacDPRSetMotors(p,4);// Update data for motors/cs 1 - 4

p  =  PMAC_DEVICE_NUMBER;
while(!DONE){
        printf("Target Position: %11.1lf\n",PmacDPRGetTargetPos(p, mtrcrd,1.0/(96.0*32.0)));
        printf("Bias Position: %11.1lf\n",PmacDPRGetBiasPos(p,mtrcrd,1.0/(96.0*32.0)));
        aDouble=PmacDPRCommanded(p,mtrcrd,'A');
        printf("Commanded Position A: %11.4lf\n",aDouble);
        aDouble=PmacDPRCommanded(p,mtrcrd,'B');
        printf("PmacCommanded Position B: %11.4lf\n",aDouble);
        aDouble=PmacDPRCommanded(p,mtrcrd,'C');
        printf("Commanded Position C: %11.4lf\n",aDouble);
        .
        .
        printf("WFE:%d\n",PmacDPRWarnFError(p,mtrcrd));
        printf("FFE:%d\n",PmacDPRFatalFError(p,mtrcrd));
        printf("AMP FAULT:%d\n",PmacDPRAmpFault(p,mtrcrd));
        printf("POSITION LIMIT:%d\n",PmacDPROnPositionLimit(p,mtrcrd));
        printf("HOME COMPLETE:%d\n",PmacDPRHomeComplete(p,mtrcrd));
        printf("MOTOR IN POSITION:%d\n",PmacDPRInposition(p,mtrcrd));
}
```

DPR Background Fixed Data Buffer Query Routines

Data Query Functions

Motor Specific

```
double PmacDPRCommanded(DWORD dwDevice,int crd,char axchar);
```

This routine returns the commanded position for an axis (described by *axchar*) in coordinate system *crd* in the same units as used to define the axis.

```
double PmacDPRVelocity(DWORD dwDevice,int motor,double units);
```

This function returns the velocity of the selected motor in units of counts per minute if the *units* parameter is set to 1.

```
double PmacDPRVectorVelocity(DWORD dwDevice,int num,int
                                              motors[], double units[]);
```

This function returns the net (or vector) velocity of the chosen *motors[]* in units of counts per minute if the *units[]* parameter is unity.

```
double PmacDPRGetTargetPos(DWORD dwDevice,int motor,
                                                    double posscale);
```

The target position of the specified *motor* is returned by this function. The units are in counts unless *posscale* is not unity.

```
double  PmacDPRGetBiasPos(DWORD dwDevice,int motor, double posscale);
```

The bias position register is returned by this function in units of counts if the *posscale* parameter is unity.

```
BOOL PmacDPRAmpEnabled(DWORD dwDevice,int motor);
BOOL PmacDPRWarnFError(DWORD dwDevice,int motor);
BOOL PmacDPRFatalFError(DWORD dwDevice,int motor);
BOOL PmacDPRAmpFault(DWORD dwDevice,int motor);
BOOL PmacDPROnPositionLimit(DWORD dwDevice,int motor);
BOOL PmacDPRHomeComplete(DWORD dwDevice,int motor);
BOOL PmacDPRInposition(DWORD dwDevice,int motor);
```

The functions above return the boolean status of a variety of motor flags.

Coordinate System Specific

```
long PmacDPRPe(DWORD dwDevice,int cs);
```

This function returns the program execution status for a given coordinate system, *cs*. The lower 24 bits are used. The low 24 bits are the same as the second word returned on a "??" command from PMAC.

```
int  PmacDPRProgRemaining(DWORD dwDevice,int cs);
```

This function returns the number of program lines remaining for a specified coordinate system, *cs*. Same as the "PR" command for PMAC.

```
long  PmacDPRTimeRemInMove(DWORD dwDevice,int cs);
```

This function returns the time remaining in the current move for a specified coordinate system, *cs*. The value this function returns is not very intuitive or useful for other than display purposes. The time for a single line in a program may be broken up in to 3 parts, acceleration, steady state, and deceleration times.

```
long  PmacDPRTimeRemInTATS(DWORD dwDevice,int cs);
```

This function returns the time remaining in accel/decl when I13>0.

```
BOOL PmacDPRRotBufFull(DWORD dwDevice,int cs);
BOOL PmacDPRSysInposition(DWORD dwDevice,int cs);
BOOL PmacDPRSysWarnFError(DWORD dwDevice,int cs);
BOOL PmacDPRSysFatalFError(DWORD dwDevice,int cs);
BOOL PmacDPRSysRunTimeError(DWORD dwDevice,int cs);
BOOL PmacDPRSysCircleRadError(DWORD dwDevice,int cs);
BOOL PmacDPRSysAmpFaultError(DWORD dwDevice,int cs);
BOOL PmacDPRProgRunning(DWORD dwDevice,int cs);
BOOL PmacDPRProgStepping(DWORD dwDevice,int cs);
BOOL PmacDPRProgContMotion(DWORD dwDevice,int cs);
BOOL PmacDPRProgContRequest(DWORD dwDevice,int cs);
```

The functions above return the boolean status of a variety of coordinate system flags.

Logical Query Functions

```
MOTIONMODE  PmacDPRGetMotionMode(DWORD dwDevice,int cs);
```

The motion mode of a chosen coordinate system is returned by this function. The MOTIONMODE enumeration is shown below:

```
typedef enum { linear,rapid,circw,circcw,spline,pvt } MOTIONMODE;
```

```
PROGRAM  PmacDPRGetProgramMode(DWORD dwDevice,int cs);
```

The program mode of a chosen coordinate system is returned by this function. The PROGRAMMODE enumeration is shown below:

```
typedef enum { stop,run,step,hold,joghold,jogstop } PROGRAM;
```

```
enum MOTION  PmacDPRGetMotorMotion(DWORD dwDevice,int motor);
```

The motion mode of a chosen motor is returned by this function. The MOTION enumeration is shown below:

```
typedef enum { inpos,jog,running,homing,handle,openloop,disabled }
MOTION;
```

DPR VARIABLE BACKGROUND READ/WRITE DATA BUFFER

PmacDPRBackground()

```
BOOL PmacDPRBackground(DWORD dwDevice, int on);
```

Starts or stops the PMAC's automatic write into the DPR background fixed and variable buffer.

Arguments

dwDevice	Device number.
on	Turn on or off Boolean.

Returns

TRUE if success.

PmacDPRBufLast()

```
int PmacDPRBufLast(DWORD dwDevice);
```

This function determines which buffer is the last variable sized buffer in DPR.

Use this function to decide which buffer (either a binary rotary or the variable background data buffer) can be initialized or removed. The sequence for the binary rotary and variable background data buffers is like a last in first out stack. The order of initialization must be: 1. Binary rotary buffer 0, 2. Binary rotary buffer 1, 3. Variable background data buffer.

Returns

From 1.16B and beyond

9 = Background Variable Buffer
8 = Binary Rotary Buffer #8
7 = Binary Rotary Buffer #7
6 = Binary Rotary Buffer #6
5 = Binary Rotary Buffer #5
4 = Binary Rotary Buffer #4
3 = Binary Rotary Buffer #3
2 = Binary Rotary Buffer #2
1 = Binary Rotary Buffer #1
0 = No Buffers in DPR

Before 1.16B

3 = Background Variable Buffer
2 = Binary Rotary Buffer #2
1 = Binary Rotary Buffer #1
0 = No Buffers in DPR

PmacDPRGetVBGAddress()

```
unsigned long PmacDPRGetVBGAddress(DWORD
   dwDevice, int handle, int entry_num);
```

This function returns an entry within the address table initialized by PmacDPRVarBufInit(). For the first entry set entry_num to 0, the second set entry_num to 1 etc. PMAC reads this address table to determine what to copy into DPR memory. Typically not used by programmers.

Arguments

dwDevice	Device number.
handle	Handle to the buffer.
entry_num	Data entry to be received from table, see for more detail

Returns

Returns the address table entry.

PmacDPRGetVBGNumEntries()

```
int  PmacDPRGetVBGNumEntries(DWORD dwDevice, int
   handle);
```

Returns the number of items being updated in the buffer by the calling application. This number is the same as that passed to the function PmacDPRVarBufInit().

Arguments

dwDevice	Device number.
handle	Handle to the buffer.

Returns

Returns the total number of entries.

PmacDPRGetVBGDataOffset()

```
UINT  PmacDPRGetVBGDataOffset(DWORD dwDevice, int
   handle);
```

Returns the offset, in bytes, from the beginning of DPR that the data for this application's VBGDB begins. Typically not used by programmers.

Arguments

dwDevice	Device number.
handle	Handle to the buffer.

Returns

Returns the offset in bytes.

PmacDPRGetVBGAddrOffset()

```
UINT  PmacDPRGetVBGAddrOffset(DWORD dwDevice, int
   handle);
```

Returns the offset, in bytes, from the beginning of DPR that the address array for this application's VBGDB begins. Divide by 4 to get the PMAC offset (from beginning of DPR 0xD000). Typically not used by programmers.

Arguments
dwDevice Device number.
handle Handle to the buffer.

Returns
Returns the offset in bytes.

PmacDPRGetVBGServoTimer()
```
UINT  PmacDPRGetVBGServoTimer(DWORD dwDevice);
```

Returns the value of the PMAC's servo timer

Arguments
dwDevice Device number.

Returns
Returns the PMAC servo timer.

PmacDPRGetVBGStartAddr()
```
UINT PmacDPRGetVBGStartAddr(DWORD dwDevice);
```

Use this to attain the PMAC address of the start of the variable data buffer. This address also indicates the end of the free user memory.

Arguments
dwDevice Device number.

Returns
The address of the start of the buffer.

PmacDPRGetVBGTotalEntries()
```
int   PmacDPRGetVBGTotalEntries(DWORD dwDevice);
```

Returns the number of items being updated in the buffer by all applications. Should never exceed 128.

Arguments
dwDevice Device number.

Returns
Returns the total number of entries.

A

PmacDPRVarBufChange()
```
long PmacDPRVarBufChange(DWORD dwDevice, long
   handle, long num_entries, long *addrarray)
```

Changes a previously initialized multi-user background variable buffer (VBGDB).

User's VBGDB structure is allocated and updated with correct address offset and data offset. If other users are present, then their VBGDB structures will also be updated.

The buffer's start address and size will be initialized along with the address array in the dual ported RAM.

Assumptions

- That this buffer was the last one to be initialized in the DPR buffer area.
- That the DPR gather size is correct modulo of the number of variables and their size (32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48-bit PMAC variable. For example have 3 variables 2 x 24 and 1 x 48 which takes 4 x 32 bits. So the size must be an exact modulo of 4 because the PMAC dual ported RAM gather does not check before storing that there is enough room remaining.
- That only one device structure is initialized per PMAC in the system

Arguments

dwDevice	Device number.
handle	Handle to previously initialized VBGDB.
num_entries	The size for address Buffer.
addrarray[]	Array of address and types.

Return Value

If it all works out a TRUE will be returned otherwise FALSE.

PmacDPRVarBufInit()

```
long PmacDPRVarBufInit(DWORD dwDevice, long
   num_entries, long *addrarray);
```

Initializes a multi-user background variable buffer. (VBGDB)

User's VBGDB structure is allocated and updated with correct address offset and data offset. If other users are present, then their VBGDB structures will also be updated.

The buffer's start address and size will be initialized along with the address array in the dual ported RAM.

Assumptions

- That this buffer was the last one to be initialized in the DPR buffer area.
- That the DPR gather size is correct modulo of the number of variables and their size (32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48 bit PMAC variable). For example have 3 variables 2 x 24 and 1 x 48 which takes 4 x 32 bits. So the size must be an exact modulo of 4 because the PMAC dual ported RAM gather does not check before storing that there is enough room remaining.
- That only one device structure is initialized per PMAC in the system

Arguments

dwDevice	Device number.
num_entries	The size for address Buffer.
addrarray[]	Array of address and types.

Return Value

A user handle if the Variable buffer is initialized OK otherwise 0.

PmacDPRVarBufRead()

```
long PmacDPRVarBufRead(DWORD dwDevice, long h,
   long entry_num, long *long_2)
```

Reads an entry from the VBGD buffer. Determines necessary offset to get the entry number desired. Returns two longs by use of a pointer **long_2*.

Calling party should poll this function until TRUE is returned. Only then will the data be valid.

Arguments

dwDevice	Device number.

h	Handle to users VBGDB status structure.
entry_num	Entry number to be returned.
long_2	Pointers to returned data. Second element may or may not be valid. It's a function of data type for entry_num in question.

Return Value

Returns TRUE(1) on data ready else FALSE(0).

PmacDPRVarBufRemove()

```
long PmacDPRVarBufRemove(DWORD dwDevice,long h);
```

Removes a user from the Multi-User Background Variable Buffer.

Arguments

dwDevice	Device number.
h	Handle to users VBGDB status structure.

Return Value

If it all works out a TRUE(1) will be returned otherwise FALSE(0).

PmacDPRWriteBuffer()

```
int PmacDPRWriteBuffer(DWORD dwDevice, int
   num_entries, struct VBGWFormat *the_data);
```

This feature is available for PROM Version 1.15G and above.

The variable background write buffer can be used to have PMAC transfer up to 32 PMAC long or short words from DPR to its own internal memory without using the DPR ASCII feature. The data to be written into PMAC's memory is first placed in an array of VBGWFormat structures (definition shown below).

```
struct VBGWFormat{
      long type_addr;
      long data1;
      long data2;
};
```

The upper 16 bits of type_addr element specifies the type of data and the lower 16 bits specify the address to which data1 and data2 should be written.

The types of data that may be specified are as follows.

Bits 0-2	0/1/2/4 for Y/L/X/SPECIAL memory respectively. For SPECIAL (PROM V 1.16 and beyond) the user must modify bits B13 and B15 for the type (0 = PLCC Ladder Diagram Function Blocks, 1..7 are for future expansion).
Bits 3-7	Width = 1,4,8,12,16,20 (a value of 0 is 24 bit variable).
Bits 8-12	Offset = 0..23.
Bits 13-15	Special type.

The way in which one uses data1, and data2 is not intuitive. For Y or X memory data1 element will be used for specifying the 24 bits of data. The most significant byte of data1 and all of data 2 are irrelevant in this case. For L data, PMAC 48 bit word, data1 should hold the first 32 bits and data2 should hold the most significant 16 bits leaving the most significant 16 bits of data2 irrelevant.

TWS is not yet implemented.

PMAC addresses are specified using an array of long integers. The most significant word of each long (upper 16 bits) specifies the word type. A value of 0, 1, 2 and 4 corresponds to Y, Long, X, and SPECIAL respectively. For Y, Long and X entries the least significant word specifies the actual PMAC address to be copied. For SPECIAL (PROM V 1.16 and beyond) the user must modify bits B13 and B15 for the type (0 = PLCC Ladder Diagram Function Blocks, 1..7 are for future expansion). When using the PLCC Ladder Diagram Function selection the least significant word specifies the function block structure member.

Arguments

num_entries	Number of elements in the the_data array to be used, max 32
the_data	A pointer to an array of VBGWFormat structures which specify what is copied from DPR into PMAC's internal memory.
entry_num	Data entry to be received from table, see above for more detail

DUAL PORTED RAM BINARY ROTARY BUFFER FUNCTIONS

PmacDPRAsciiFileToRot()

```
SHORT PmacDPRAsciiFileToRot(DWORD dwDevice, FILE
   *inpfile, USHORT bufnum);
```

Inputs ASCII strings from file "*inpfile*", converts all alpha to upper case, then converts the ASCII to PMAC commands and writes the data to the dual ported RAM binary rotary buffer when ready.

Arguments

dwDevice	Device number.
inpfile	NULL terminated path to file.
bufnum	Binary rotary buffer number.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons, when this occurs the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRAsciiStrToBinaryFile()

```
SHORT PmacDPRAsciiStrToBinaryFileA(PCHAR inpstr,
   FILE *outfile);

SHORT PmacDPRAsciiStrToBinaryFileW(PWCHAR inpstr,
   FILE *outfile);
```

Inputs ASCII string then converts the ASCII to PMAC commands and writes the binary data to the "*outfile*". The file is not closed.

On a ASCII to binary PMAC command conversion error, the ‘C’ runtime library function fgetpos() can be used to get the end of line byte position of where the error occurred.

Arguments

dwDevice	Device number.
inpstr	NULL terminated string.
outfile	FILE structure handle of output file.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons, when this occurs the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRAsciiStrToRot()

```
SHORT PmacDPRAsciiStrToRotA(DWORD dwDevice, PCHAR
   inpstr, USHORT bufnum)

SHORT PmacDPRAsciiStrToRotW(DWORD dwDevice,
   PWCHAR inpstr, USHORT bufnum)
```

PmacDPRAsciiStrToRot() takes an ASCII Native PMAC text string, converts it to Native PMAC Binary, then places it into the DPR Binary Rotary Buffer if it has been setup and there is room.

Arguments

dwDevice	Device number.
inpstr	NULL terminated PMAC command string.
Bufnum	Binary rotary buffer number.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons, when this occurs the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRAsciiToBinaryFile()

```
SHORT PmacDPRAsciiToBinaryFile(FILE *inpfile,FILE
    *outfile);
```

Inputs ASCII string from file "*inpfile*", converts all alpha to upper case, then converts the ASCII to PMAC commands and writes the binary data to the "*outfile*". Neither file is closed.

On a ASCII to binary PMAC command conversion error, the 'C' runtime library function fgetpos() can be used to get the end of line byte position of where the error occurred.

Arguments

dwDevice Device number.
inpfile NULL terminated path of input file.
outfile NULL terminated path of output file.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons, when this occurs the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRBinaryFileToRot()

```
SHORT PmacDPRBinaryFileToRot(DWORD dwDevice, FILE
    *inpfile, USHORT bufnum);
```

Inputs Binary PMAC commands from file "*inpfile*" and writes the data to the DPR binary rotary buffer when ready.

Arguments

dwDevice Device number.
inpfile NULL terminated path to file.
bufnum Binary rotary buffer number.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons, when this occurs the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRBinaryToRot()

```
SHORT PmacDPRBinaryToRot(DWORD dwDevice, WORD
   *inpbinptr,  WORD **outbinptr, WORD bufnum );
```

Inputs Binary PMAC commands from RAM and writes the data to the DPR binary rotary buffer when ready.

Arguments

dwDevice	Device number.
Inpbinptr	Pointer to input binary PMAC commands RAM.
outbinptr	Pointer to store where the next read should come from.
Bufnum	Binary rotary buffer number.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons, when this occurs the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRRotBufInit()

```
int  PmacDPRRotBufInit(DWORD dwDevice, int
   bufnum);
```

The PmacDPRRotBufInit() function is used to initialize the DPR Binary Rotary motion program buffers.

Presently, there may be up to 8 rotary buffers, buffer #0, buffer #1 etc. Buffers #0, and #1 will always use coordinate system 1 and 2 respectively. The size of the buffers is configurable with the registry - variables DualPortRamRot1 and DualPortRamRot2 etc.

The size of a particular buffer may also be changed after PMAC driver initialization takes place by calling the () function. Note that using PmacDPRRotBufChange() will not cause the change in size to be persistent when resetting the operating system (registry is not changed). Note you should set I57 to 0 before changing the buffer size, and the buffer being changed should be the one last initialized.

The size of the rotary buffer in PMAC is typically half that allocated in the DP Ram. Two long words in the DP Ram, again, is mapped into 1 long PMAC word (48 bit).

The value assigned to DualPortRamRotx (where *x* = 1,2), which is in units of 32 bits, specifies how many instructions will fit into the buffer. Each ASCII instruction is ultimately transformed into a 64-bit code (from which PMAC will transform into a 48 bit operation code). A single instruction consists of a program command and it's argument. For example "X100" is a single instruction and "X1000Y1000Z1000" is 3 instructions. A simple formula to determine the bufsize as a function of the maximum number of buffered program statements is:

buffer_size = (number_of_motors_in_cs) * (num_lines) * (2);

Call () to enable the Binary Rotary Buffer after it's been initialized.

Arguments

dwDevice	Device number.
bufnum	Which of the two rotary buffers (i.e. 0 or 1) to reference.

Non-Parameter Inputs

Buffer Size

Registry value "DualPortRamRot(bufnum)" = the size for the desired buffer in (32 bit word units).

If the value of this key in the registry is 0 this effectively removes this rotary buffer. If greater than 0 this means you want to initialize #1, #2 etc. in DPR. The order must be rotary buffer #1 first then #2 etc.

If the size has not changed, this will flush the buffer.

You can resize this buffer by calling "PmacDPRRotBufChange" however, it must be the last buffer in DPR. To determine which was the last allocated buffer use PmacDPRBufLast().

Notes

Variable sized buffers such as this one are added to the end of available buffer memory. This leaves room for the Gather buffer which is fixed at the beginning of the DPR address space.

Assumptions

That the DPR gather size is a correct modulo of the number of variables and their size (32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48 bit variable).

Return Values

RET_ERROR (-1) (Check request, the buffer size was either too small or too large)
RET_OK (0)

PmacDPRRotBufRemove()

```
int    PmacDPRRotBufRemove(DWORD dwDevice, int
   bufnum);
```

Use this function to remove any previously initialized rotary buffer.

Arguments

dwDevice	Device number.
Bufnum	Which of the two rotary buffers (i.e. 0 or 1) to reference.

Return Values

RET_ERROR (-1) (Check request, the buffer size was either too small or too large)

RET_OK (0)

PmacDPRRotBufChange()

```
int PmacDPRRotBufChange(DWORD dwDevice, int
   bufnum,int new_size) ;
```

Use this function to change the size of a rotary buffer. This will remove the existing rotary buffer, and allocate a new one in DPR. If size is set to zero, this function is equivalent to ().

The size of a particular buffer may be changed after PMAC driver initialization takes place by calling this function. Note that using PmacDPRRotBufChange() will not cause the change in size to be persistent when resetting the operating system (registry is not changed). You must change the registry to be persistent.

Presently, there may be up to 8 rotary buffers, buffer #0, buffer #1 etc. Buffers #0, and #1 will always use coordinate system 1 and 2 respectively. The size of the buffers is configurable with the registry - keys "DualPortRamRot1" and "DualPortRamRot2" etc.

The size of the rotary buffer in PMAC is typically half that allocated in the DPR. Two long words in the DP Ram, again, are mapped into 1 long PMAC word (48 bit).

> Note I57 should be set to 0 before changing the buffer size, and the buffer being changed must be the last variable sized buffer initialized.

Arguments

dwDevice	Device number.
bufnum	Which of the two rotary buffers (i.e. 0 or 1) to reference.
new_size	Size of new buffer.

Return Values

RET_ERROR (-1) Error (Check request, the buffer size was either too small or too large)
RET_OK (0)

PmacDPRRotBufClr()

```
void  PmacDPRRotBufClr(DWORD dwDevice, int
   bufnum);
```

This function will clear either Binary Rotary buffers 0 or 1 in DPR (i.e. remove all entries).

Arguments

dwDevice	Device number.
bufnum	Which of the two rotary buffers (i.e. 0 or 1) to reference.

Return Values

None

PmacDPRRotBuf()

```
int    PmacDPRRotBuf(DWORD dwDevice,BOOL onoff);
```

Once initialized with () the PmacDPRotBuf() function can be used to enable or disable the rotary buffer (if **onoff** = 1 then enable if 0 then disable). The return value is a Boolean, TRUE for enabled, FALSE for disabled.

Arguments

dwDevice	Device number.
onoff	Boolean value, Use 1 (ON) or 0 (OFF).

Return Values

1 = Active, 0 = Non-Active

DPR NUMERIC READ AND WRITE

General Information

The PmacDPRSet{DataType}() functions write numerical data to the specified **offset** while the PmacDPRGet{DataType}() reads at the specified **offset** and returns the data.

From within PMAC, data can be written to the DPR by use of PMAC M-variable assignments. Proper M-variable definitions for the corresponding data type are shown below:

Data Type	M variable Definition
16 bit integer	M{constant}>X/Y:{Address} (i.e. m100->X:$D200,0,16,s)
32 bit integer	M{constant}->DP:{Address} (i.e. m101->DP:$D201)
32 bit floating point	M{constant}->F:{Address} (i.e. m102->DP:$D202)

Standard Read/Write

```
WORD PmacDPRGetWord(DWORD dwDevice,UINT offset);
void  PmacDPRSetWord(DWORD dwDevice,UINT offset,
   WORD val);
DWORD PmacDPRGetDWord(DWORD dwDevice,UINT
   offset);
void  PmacDPRSetDWord(DWORD dwDevice,UINT
   offset, DWORD val);
float PmacDPRGetFloat(DWORD dwDevice,UINT
   offset);
void  PmacDPRSetFloat(DWORD dwDevice,UINT
   offset,float val);
```

Masking

```
BOOL PmacDPRDWordBitSet(DWORD dwDevice,UINT
   offset,UINT bit);
void PmacDPRSetDWordBit(DWORD dwDevice,UINT
   offset,UINT bit);
void PmacDPRResetDWordBit(DWORD dwDevice,UINT
   offset,UINT bit);
void PmacDPRSetDWordMask(DWORD dwDevice,UINT
   offset,DWORD val,BOOL onoff);
DWORD PmacDPRGetDWordMask(DWORD dwDevice,UINT
   offset,DWORD val);
```

Dual Word Conversion

```
double PmacDPRFloat(DWORD dwDevice,long
   d[],double scale);
double    PmacDPRLFixed(DWORD dwDevice,long
   d[],double scale);
```

The "Dual Word" Conversion functions convert data that is placed in DPR by one of its automatic features. Whenever a long word in PMAC (48 bit) is placed in DPR (Motor 1 actual position register for example) each 24 bit short word (X and Y) is sign extended and placed in a 32-bit word making it 64 bits of data that need to be converted.

A typical sequence of function calls for retrieving and converting motor 1 actual position for example would look like:

```
/* Get the dual word data */
long data[2];

While(1){
        /* Get second entry*/
        if(PmacDPRVarBufRead(dwDevice,myVBGBHandle,1,data))
        {
             /* Convert to encoder counts (Assuming I108 = 96)*/
             MotorPosition =
PmacDPRLFixed(dwDevice,data,1.0/(96.0*32.0));
             cprintf("\n\n Actual motor position after conversion
        =%lf\r\n",MotorPosition);
        }
}
```

The second parameter passed to PmacDPRLFixed() is the scale factor. Since this position register in units of (32*I108) encoder counts the inverse of this is used to scale the return value.

PmacDPRDWordBitSet()

```
BOOL PmacDPRDWordBitSet(DWORD dwDevice, UINT
   offset, UINT bit);
```

Determines if a bit is set within a 32-bit word (DWORD) in dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
bit	Bit number to set. 0..31.

Return Values

TRUE if bit == 1.

PmacDPRFloat()

```
double PmacDPRFloat(long d[],double scale);
```

Converts a 48-bit floating point word: 36-bit mantissa / 12 exponent packed into two 32-bit words holding 24 bits each of the 48 bit words. Multiplies the result by "*scale*".

Arguments

d[ ]	Two 32 bit long values to converts.
scale	Scale multiplier.

Return Values

Double floating point representation of the 48-bit number.

PmacDPRGetDWord()

```
DWORD PmacDPRGetDWord(DWORD dwDevice,UINT
   offset);
```

Retrieves a 32-bit DWORD from dual ported RAM.

Arguments

dwDevice	Device number.
Offset	Offset from the start of dual ported RAM.

Return Values

DWORD at offset location.

PmacDPRGetDWordMask()

```
DWORD  PmacDPRGetDWordMask(DWORD dwDevice, UINT
   offset, DWORD mask);
```

Retrieves a double word from dual ported RAM memory at the **offset**, masks (bitwise AND) it with **val** and then returns the resulting value.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
mask	Bitwise and mask used with value read from dual ported RAM.

Return Values

Result of DWORD and mask at offset location.

PmacDPRGetFloat()

```
float  PmacDPRGetFloat(DWORD dwDevice, UINT
   offset);
Reads an IEEE 32-bit floating point value from
   dual ported RAM.
```

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.

Return Values

Float at offset location.

PmacDPRGetMem()

```
PVOID PmacDPRGetMem(DWORD dwDevice, DWORD offset,
   size_t count, PVOID val);
```

Copies a block of dual ported RAM memory.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
count	Size of memory block to copy.
val	Pointer to destination.

Return Values

Pointer to destination.

PmacDPRGetWord()

```
WORD PmacDPRGetWord(DWORD dwDevice, UINT offset);
```

Retrieves a 16-bit word from dual ported ram.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.

Return Values

Pointer to destination.

PmacDPRResetDWordBit()

```
void PmacDPRResetDWordBit(DWORD dwDevice, UINT
   offset, UINT bit);
```

Exclusive XORS a bit within a 32-bit word (DWORD) in dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
bit	Bit number to set. 0..31.

Return Values

Nothing.

PmacDPRSetDWord()

```
void PmacDPRSetDWord(DWORD dwDevice, UINT offset,
   DWORD val);
```

Sets the value of a 32-bit word in dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
val	Value to write to DPRAM.

Return Values

Nothing.

PmacDPRSetDWordBit()

```
void PmacDPRSetDWordBit(DWORD dwDevice, UINT
   offset, UINT bit);
```

Turns on a bit (sets to one(1)) within a 32-bit word (DWORD) in dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
bit	Bit number to set. 0..31.

Return Values

Nothing.

PmacDPRSetDWordMask()

```
void PmacDPRSetDWordMask(DWORD dwDevice, UINT
   offset, DWORD val, int onoff);
```

This function is used to manipulate DPR memory a DWORD (32 bits) at time. A bit wise logical OR of the register specified by **offset** is done with **val** and then stored into the same specified DPR location**.**. If **onoff** is TRUE the a bit wise exclusive or XOR is evaluate using **val** and the specified memory location. The result of this operation is then used to perform a bit wise AND with the value located in DPR. The result is stored in the specified DPR location.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
val	Value to store.
onoff	Boolean indicating exclusive XOR or AND of *val* with DPRAM memory.

Return Values

Nothing.

PmacDPRSetFloat()

```
void PmacDPRSetFloat(DWORD dwDevice, UINT offset,
   double val);
```

Writes a floating point value into dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
val	Value to store.

Return Values

Nothing.

PmacDPRSetMem()

```
PVOID PmacDPRSetMem(DWORD dwDevice, DWORD offset,
   size_t count, PVOID val);
```

Copies a block of memory into dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
count	Size of data to transfer.
val	Pointer to memory to transfer.

Return Values

Pointer to transferred data.

PmacDPRSetWord()

```
void PmacDPRSetWord(DWORD dwDevice, UINT offset,
   WORD val);
```

Writes a 16-bit word value into dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
val	Value to store.

Return Values

Nothing.

INTERRUPT FUNCTIONS

General Information Interrupt Functions

Interrupts are provided for both Windows 95/98 and Windows NT operating systems. There are four methods of interrupt notification for your program:

1. Send a message to a window.
2. Call a function.
3. Start a thread function.
4. Set an event.

All four methods have a separate initialization function. Common to all initialization functions is the parameter *ulMask*. This parameter determines the interrupt service vector(s) to be used for the interrupt initiated by the function.

The least significant byte of *ulMask* controls which conditions will generate an interrupt. A bit value of 0 enables, 1 disables.

Bit	PMAC Signal
0	In Position of Coordinate System
1	Buffer Request (PMAC's request for more moves)
2	Error, A motor(s) in the coordinate system has had a fatal following error
3	Warning, A motor(s) in the coordinate system has had a warning following error
4	Host Request, PMAC has an ASCII response for the host
5-7	User programmable, see PMAC User's Guide, Writing a Host Communications Program

PmacINTRFireEventInit()

```
BOOL  PmacINTRFireEventInit(DWORD dwDevice,HANDLE
   hEvent,ULONG ulMask);
```

Once initialized with PmacINTRFireEventInit() an interrupt will cause the driver to set the event handle provided to the signaled state. The programmer is responsible to reset the event to the non-signaled state.

Arguments

dwDevice	Device number.
hEvent	Handle to the event to be set.
ulMask	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

PmacINTRFuncCallInit()

```
BOOL  PmacINTRFuncCallInit(DWORD
   dwDevice,PMACINTRPROC pFunc,DWORD msg,ULONG
   ulMask);
```

Once initialized with PmacINTRFuncCallInit() an interrupt will cause the driver to call a function provided of PMACINTRPROC type.

Arguments

dwDevice	Device number.
pFunc	Pointer to a PMACINTRPROC function.
msg	Message to be passed back to the called function as a parameter.
ulMask	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

PmacINTRRunThreadInit()

```
BOOL  PmacINTRRunThreadInit(DWORD
   dwDevice,LPTHREAD_START_ROUTINE pFunc,UINT
   msg,ULONG ulMask);
```

Once initialized with PmacINTRRunThreadInit() an interrupt will cause the driver to initialize a thread function provided of LPTHREAD_START_ ROUTINE type. The programmer is responsible for synchronization and termination of the thread.

Arguments

dwDevice	Device number.
pFunc	Pointer to a LPTHREAD_START_ROUTINE function.
msg	Message to be passed back to the called function as part of a parameter structure.
ulMask	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

PmacINTRTerminate()

```
BOOL  PmacINTRTerminate(DWORD dwDevice);
```

Shuts down the interrupt service.

Arguments

dwDevice	number.

Return Values

TRUE = Ok, FALSE = Error.

PmacINTRWndMsgInit()

```
BOOL  PmacINTRWndMsgInit(DWORD dwDevice,HWND
   hWnd,UINT msg,ULONG ulMask);
```

Once initialized with PmacINTRWndMsgInit() an interrupt will cause the driver to send provided message to the provided windows handle.

Arguments

dwDevice	Device number.
hWnd	Window handle.
msg	Message to be sent to the window.
ulMask	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

Serial Checksum Communication Functions

```
int PmacSERDoChecksums(DWORD dwDevice, UINT
   do_checksums);

int PmacSERCheckResponseA(DWORD dwDevice, char *
   response, UINT maxchar,char * outchar);

BOOL PmacSERCheckSendLineA(DWORD dwDevice,char *
   outchar, char * command_csum);

int PmacSERCheckGetLineA(DWORD dwDevice,PCHAR
   response,UINT maxchar, PUINT num_char);
```

Note:

See Terminal.C example program for a terminal example of using checksums.

PmacSERDoChecksums()

To enable or disable serial checksummed communications call PmacSERDoChecksums(). A non-zero return value indicates a successful mode change.

Once enabled, the last three routines above may be called. By far the easiest and most useful is the PmacSERCheckResponseA() routine.

PmacSERCheckResponseA()

PmacSERCheckResponseA() is used to provide checksum verification of commands sent to PMAC. It is capable of retrieving both multiple and single line responses. This function returns the total number of characters received from PMAC including checksum characters. Since the checksum characters are removed from the response, the final number of characters in the response, and the value return by this function will differ. In any event, if the value returned by this function becomes equal to the "maxchar" parameter PMAC's total response was not received. If this occurs, use SERflush to clear out PMAC's buffer, increase the size of the character array (and also maxchar) and try again. If an error occurs this function will return FALSE.

This function uses the "control-n" feature implemented in PROM versions 1.14 and beyond. This feature enables the host to assure that PMAC has received the command correctly before sending the carriage return which tells PMAC to act on the just sent command.

NOTE: If PMAC's response is larger than "maxchar", the remainder of the response will be left in the rotary buffer. This can cause garbled communications, therefore, always oversize the "response" buffer and the "maxchar" specification

The following error codes may be set after calling this routine:

IDS_CHECKSUMDATABAD	Checksum communications error Bad Data.
IDS_CHECKSUMACKBAD	Checksum communications error Bad Acknowledge

See below for the source of this routine.

PmacSERCheckSendLineA()

Function: This function sends a line to PMAC but first checks to see if it got down correctly. It returns TRUE on successful transmission of the command. The command sum for the command is also passed back by value. See below for an example of it's usage within the PmacSERCheckResponseA() routine.

PmacSERCheckGetLineA()

This function is used to retrieve and checksum verify a response from PMAC. For multiline responses this routine pulls out one line at a time. A line is delimited by a CR. It checks each lines Data Checksum. When the ACK comes at the end, the command checksum is placed in *response[0]* and COMM_EOT is returned.

If either a checksum, or a timeout occurs, COMM_FAIL is returned and the appropriate error code set (call GetError(), GetErrorStr() to retrieve the error). See below for an example of it's usage within the PmacSERCheckResponseA() routine.

PmacSERCheckResponse() Source/Example Error! Bookmark not defined.

int PmacSERCheckResponse(DWORD dwDevice,char * response, UINT maxchar,char * outchar)

```
{
      char *cp, csum, command_sum_from_PMAC;
      DWORD nc =0;
      int     ic,attempt,ret;
      DWORD tc;
  response[0]=0;
  si.SERCommError     = ERR_OK;
  attempt = 0;
  if(si.Initialized == FALSE) return 0;
  if(strlen(outchar) == 0)
    return 0;
  while(attempt < NUM_CHECKSUM_RETRIES){
    if(outchar[0] <= 26)// It's a control char
    {
      csum = outchar[0];
      PmacSERSendCharA(dwDevice,outchar[0]);
    }
    else if(!PmacSERCheckSendLine(dwDevice,outchar,&csum))
      goto error;
    cp = response;
        tc = maxchar;
        while(tc > 0 && (ret = PmacSERCheckGetLine(dwDevice,cp,tc,&ic))
>= 0)
    {
            tc -= (ic +1);
              cp += ic;
    }
```

```
    if(IS_COMM_FAIL(ret))
    {
    //          si.SERCommError = IDS_CHECKSUMDATABAD;     // compare
data checksum
           //    si.SERCommError = COMM_TIMEOUT;
      goto error;
    }
    if(IS_COMM_EOT(ret))
    {
      command_sum_from_PMAC = *cp;
      *cp = 0;// Remove the checksum from the response.
      if(command_sum_from_PMAC == !csum)
      { // IS IT AN ACKNOWLEDGE
        si.SERCommError = IDS_CHECKSUMACKBAD;
               goto error;
     }
    }
    if(IS_COMM_ERROR(ret))
                goto error;
    if(nc == maxchar)
    {
                si.SERCommError = IDS_INCOMP_RECEIVE; // Unable to get
complete response
                goto error;
           }
// Remove checksums from response
   nc = (maxchar - tc);
       return nc;
error:
           attempt++;
           SERFlush(dwDevice);
     }
     return 0;
}
```

VARIABLE FUNCTIONS

PmacGetVariable()

```
short int PmacGetVariable(DWORD dwDevice,char
   ch,UINT num,short int def);
long PmacGetVariableLong(DWORD dwDevice,char
   ch,UINT num,long def);
double PmacGetVariableDouble(DWORD dwDevice,char
   ch,UINT num,double def);
```

Returns a PMAC variable (I, M, P or Q) value as a 16-bit integer, long 32-bit integer or 80-bit floating point double. If there is a communication error then the default value in parameter 'def' will be returned.

Arguments

dwDevice	Device number.
Ch	Character representing the variable type. 'M', 'P', 'I', 'Q'.
num	Variable number to get.
def	Default value.

Return Value

'I' variable value or default.

PmacSetVariable()

```
void PmacSetVariable(DWORD dwDevice, char ch,
   UINT num, short int val);
void PmacSetVariableLong(DWORD dwDevice, char ch,
   UINT num, long val);
void PmacSetVariableDouble(DWORD dwDevice, char
   ch, UINT num, double val);
```

Set a PMAC variable value as a 16bit integer, long 32bit integer or 80bit floating point double.

Arguments

dwDevice	Device number.
Ch	Character representing the variable type. 'M', 'P', 'I', 'Q'.
num	Variable number.
val	Value to be set.

Return Values

Nothing.

PmacSetIVariable()

```
void PmacSetIVariable(DWORD dwDevice, UINT num,
   short int val);
void PmacSetIVariableLong(DWORD dwDevice, UINT
   num, long val);
void PmacSetIVariableDouble(DWORD dwDevice, UINT
   num, double val);
```

Set a PMAC 'I' variable value as a 16bit integer, long 32bit integer or 80bit floating point.

Arguments

dwDevice	Device number.
num	'I' variable number.
val	Value to be set.

Return Values

Nothing.

PmacGetIVariable()

```
short int PmacGetIVariable(DWORD dwDevice,UINT
   num,short int def);
long PmacGetIVariableLong(DWORD dwDevice,UINT
   num,long def);
double PmacGetIVariableDouble(DWORD dwDevice,UINT
   num,double def);
```

Returns a PMAC 'I' variable value as a 16bit integer, long 32bit integer or 80bit floating point. If there is a communication error then the default value in parameter 'def' will be returned.

Arguments

dwDevice	Device number.
num	'I' variable number to get.
def	Default value.

Return Value

'I' variable value or default.

UTILITY FUNCTIONS

PmacGetRomDateA()

```
PCHAR PmacGetRomDateA(DWORD dwDevice,PCHAR s,int
   maxchar);
```

Fills a string buffer with the firmware date of the PMAC device.

Arguments

dwDevice	Device number.
s	Pointer to string buffer.
Maxchar	Maximum characters to copy.

Return Value

Pointer to string buffer.

PmacGetRomVersionA()

```
PCHAR PmacGetRomVersionA(DWORD dwDevice,PCHAR
   s,int maxchar);
```

Fills a string buffer with the firmware version of the PMAC device.

Arguments

dwDevice	Device number.
s	Pointer to string buffer.
Maxchar	Maximum characters to copy.

Return Value

Pointer to string buffer.

PmacGetUserHandle()

```
PUSER_HANDLE PmacGetUserHandle(DWORD dwDevice);
```

This function is used to retrieve a handle to a communication information structure used internally by the driver. It is very similar to the PCOMM Library tagPmac structure. You will probably never need to use it.

Arguments

dwDevice	Device number.

Return Value

NULL if the function fails. Otherwise a pointer to USER_HANDLE.

PmacGetPmacType()

```
int PmacGetPmacType(DWORD dwDevice);
```

Returns an integer indicating the PMAC model you are communicating with.

Arguments

dwDevice	Device number.

Return Value

1 = PMAC1.
2 = PMAC2.

PmacGetIORegister()

BYTE PmacGetIORegister(DWORD dwDevice,BYTE offset)

Allows the user to get at PMAC's I/O registers, (BUSBASE + x). Useful for checking PMAC state without having to use ASCII communications. Other utility functions are provided which do similar things, but use ASCII communications.

Arguments

dwDevice Device number.

Offset Offset from PMAC's bus base address.

Return Value

PMAC's Interrupt Status Register State if successful

0xFF if a failure occurred, either communication not established or BUS or DPR communication is not being used

Assumptions

1. Either the bus or DPR is being used. No good for serial port.
2. If checking InPosition or other similar data bits from the status register (base + 2), the appropriate Motor or Coordinate system has been addressed.

PmacSetIORegister()

BOOL PmacSetIORegister(DWORD dwDevice,BYTE offset,BYTE value){

Allows the user to set PMAC's I/O registers, (BUSBASE + x). Useful for getting PMAC out of a reset state for example. Either the Bus or DPR must be in use for this routine to succeed

RETURNS:
TRUE if successful, else FALSE

Arguments

dwDevice Device Number (PMAC 1,2,3 etc.)

offset Offset from PMAC's bus base address

value value to be written to port

Return Value

TRUE if successful, else FALSE

Assumptions

Either the bus or DPR is being used. No good for serial port.

getBitValue()

```
int getBitValue(char *hexadecimal_number, int
   bit_requested);
```

Gets the value of a bit from a hexadecimal string.

Arguments

hexadecimal_number Zero terminated hex string.

bit_requested `Bit number to examine.`

Return Value

One(1) or zero(0).

hex_long2()

```
long hex_long2(char *in_str, int str_ln);
```

Converts a hexadecimal string to a long integer.

Arguments

in_str	Pointer to the beginning of hex string.
str_ln	How many characters to examine.

Return Value

Long integer value representation of hex string.

DATA TYPES, STRUCTURES, CALLBACKS, AND CONSTANTS

DOWNLOADGETPROC

```
typedef int  (FAR WINAPI *DOWNLOADGETPROC)(int
   nIndex,LPTSTR lpszBuffer,int nMaxLength);
```

This function type is used by some program downloading procedures to offer the option of extracting text lines for the download from another source other than a disk file.

Arguments

nIndex	Line number asked for.
lpszBuffer	Buffer pointer to copy text line into.
nMaxLength	Maximum length of buffer.

Return Value

The number of characters copied into the buffer.

DOWNLOADPROGRESS

```
typedef void (FAR WINAPI *DOWNLOADPROGRESS)(int
   nPercent);
```

This function type is used by some program downloading procedures to offer the option of setting the current percent of progress during the procedure.

Arguments

nPercent	Current percent of progress.

Return Value

Nothing.

DOWNLOADMSGPROC

```
typedef void (FAR WINAPI *DOWNLOADMSGPROC)(LPTSTR
   str,BOOL newline);
```

This function type is used by some program downloading procedures to offer the option of reporting a status message.

Arguments

str	Message string.
newline	Indicates if a new line should be added by the called procedure.

Return Value

Nothing.

DPRTESTMSGPROC

```
typedef void (FAR WINAPI *DPRTESTMSGPROC)(LONG
   NumErrors,LPTSTR Action,LONG CurrentOffset);
```

This function type is used by some program testing DPR via the PmacDPRTest() routine to offer the option of reporting the number of errors that have occurred, a status message, and the current offset being tested. The *action* message indicates to the user what action is actively being done in the testing of the DPR. The *action* message will also indicate to the user what errors occurred during the testing.

Action
"Setting up for Dual Ported Ram Test"
"Cannot set communications mode to BUS."
"Writing to DPR address (via PMAC)"
"Reading/Verifying DPR (via PC)"
"Writing to DPR Address (via PC)"
"Reading/Verifying DPR (via PMAC)"
"Clearing all Dual Ported Ram"

Arguments

NumErrors	Number of accumulated errors that have occurred
Action	Message indicating what part of test is being done
CurrentOffset	Offset from host base address being tested (in bytes)

Return Value

Nothing.

DPRTESTPROGRESS

```
typedef void (FAR WINAPI *DPRTESTPROGRESS)(LONG
   Percent);
```

This function type is used by some program testing DPR via the PmacDPRTest() routine to offer the option of reporting a status message. The message indicates to the user what percent the test procedure is done with the current action being done. Actions include the following:

Action
"Setting up for Dual Ported Ram Test"
"Cannot set communications mode to BUS."
"Writing to DPR address (via PMAC)"
"Reading/Verifying DPR (via PC)"
"Writing to DPR Address (via PC)"
"Reading/Verifying DPR (via PMAC)"
"Clearing all Dual Ported Ram"

Arguments

Percent	Percent done with current action being done in test

Return Value

Nothing.

Dpr_Bin_Rot_Errors

Used by binary rotary buffer functions.

```
enum Dpr_Bin_Rot_Errors
{
  DprOk          =   0,
  DprBufBsy      =   1,      // DPR Rotary Buffer is Busy
  DprEOF         =   2,      // DPR Rotary Buffer End of File detected
  DprFltErr      =  -1,      // Unable to pack; floating point number
                               too large
  DprStrToDErr   =  -2,      // Unable to convert string to double float
                               number
  DprMtrErr      =  -3,      // Unable to convert Motor numbers
                               specified
  DprPlcErr      =  -4,      // Unable to convert PLC numbers specified
```

```
  DprAxisErr    =  -5,     // Unable to convert axis in ABS, INC or
                      FRAX
  DprCmdErr     =  -6,     // Illegal Command in string
  DprCmdMaxErr  =  -7,     // Exceeded the Max number of PMAC cmds in
                      a line  ( See DPRCMDMAX )
  DprIntErr     =  -8,     // Integer number out of range ( See
                      Integer Max values CIRMAX, etc. )
  DprBufErr     =  -9,     // DPR Rotary or Internal Rotary Buffer
                      Size is zero
  DprOutFileErr = -10,     // DPR Output File Error
  DprInpFileErr = -11 // DPR Input File Error
};
```

GLOBALSTATUS

Used in DPR Real Time Buffer query routines

```
typedef struct gs { // Global Status
     // DWord 1 ( ??? 1st 24/32 bit word )
     USHORT rffu2                  : 8; // 0-7
     USHORT internal1      : 3; // 8-10
     USHORT buffer_full       : 1;
     USHORT internal2      : 3; // 12-14
     USHORT dpram_response : 1;
     USHORT plc_command       : 1;
     USHORT plc_buf_open   : 1;
     USHORT rot_buf_open   : 1; // 18
     USHORT prog_buf_open  : 1; // 19
     USHORT internal3      : 2;
     USHORT host_comm_mode : 1;
     USHORT internal4      : 1;
     USHORT pad2                   : 8;
     // DWord 2 ( ??? 2nd 24/32 bit word )
     USHORT rffu1                  : 7;
     USHORT end_gather           : 1;
     USHORT rapid_m_flag   : 1;
     USHORT rti_warning       : 1;
     USHORT earom_error       : 1;
     USHORT dpram_error       : 1;
     USHORT prom_checksum  : 1;
     USHORT mem_checksum   : 1;
     USHORT comp_on           : 1;
     USHORT stimulate_on   : 1;
     USHORT stimulus_ent   : 1;
     USHORT prep_trig_gat  : 1;
     USHORT prep_next_serv : 1;
     USHORT data_gat_on       : 1;
     USHORT servo_err      : 1;
     USHORT servo_active   : 1;
     USHORT intr_reentry   : 1;
     USHORT intr_active       : 1;
     USHORT pad1                   : 8;
} GLOBALSTATUS;
```

MOTION

typedef enum { inpos,jog,running,homing,handle,openloop,disabled } MOTION;

MOTIONMODE

typedef enum { linear,rapid,circw,circcw,spline,pvt } MOTIONMODE;

PROGRAM

typedef enum { stop,run,step,hold,joghold,jogstop } PROGRAM;

SERVOSTATUS

Used in DPR Real Time Buffer query routines

```
typedef struct ss { // Motor Servo Status ( ?  1st 24 bit word)
      USHORT internal1      : 8;
      USHORT internal2      : 2;
      USHORT home_search    : 1;
      USHORT block_request  : 1;
      USHORT rffu1          : 1;
      USHORT desired_vel_0  : 1;
      USHORT data_block_err : 1;
      USHORT dwelling       : 1;
      USHORT integration    : 1;
      USHORT run_program    : 1;
      USHORT open_loop      : 1;
      USHORT phased_motor   : 1;
      USHORT handwheel_ena  : 1;
      USHORT pos_limit      : 1;
      USHORT neg_limit      : 1;
      USHORT activated      : 1;
      USHORT pad            : 8;
} SERVOSTATUS;
```

EXTENDED FUNCTIONS

The following functions are extended version of the existing functions (without the extension "**Ex**"). The "**Ex**" signifies the enhancement to the legacy code by their ability to get the actual status bit from the PMAC in addition to the number of characters. Furthermore, these functions have eliminated the existing bugs associated with the legacy code.

The purpose of using these new functions is to get the actual status code instead of the number of characters as was the case of legacy code (without Ex). The main difference from the old functions to the new ones is the return value. In the existing code the number of characters is returned. The new functions are written to get the error status in addition if any and are explained in detail in the following section.

It is strongly recommended that users use these new functions instead of the old functions.

PmacGetLineExA

PmacGetLineExW

PmacGetLineEx*X*()

```
int PmacGetLineEx A(DWORD dwDevice, PCHAR linebuf,
   UINT maxchar);

int PmacGetLineEx W(DWORD dwDevice, PWCHAR
   linebuf, UINT maxchar);
```

PmacGetLineEx*X*() queries PMAC for a line response. This function will wait for one of the following conditions to occur before returning:

1. a *timeout* period
2. PMAC has a response (A <CR> (ASCII 13) or an <ACK> (ASCII 6) was received from PMAC)
3. The maximum number of characters, maxchar, was received

If PmacGetLineEx*X*() does not succeed on its first attempt to retrieve the response it may be that the PMAC device has no response available or a Timeout has occurred waiting for PMAC to respond. You can use the function () to first determine if a response is available.

Arguments

dwDevice	Device number.
linebuf	Pointer to character buffer to receive character string from PMAC.
maxchar	Maximum number of characters to be received from PMAC.

Return Value

Number of characters received along with the status bits (including handshake characters).

FALSE(0) One of the following occurred:

1. No characters in PMAC's buffer queue
2. An error occurred
3. Communication not established

PmacGetResponseExA
PmacGetResponseExW

PmacGetResponseEx*X*()

```
int PmacGetResponseExA(DWORD dwDevice, PCHAR response,
      UINT maxchar, PCHAR command);

int    PmacGetResponseExW(DWORD    dwDevice,    PWCHAR
      response, UINT maxchar, PWCHAR command);
```

Most if not all of your communication with the PMAC can be handled with the PmacGetResponseEx*X*() function. This function will send a command string (i.e. "#1j+, "?", "Open Prog1", etc.) to the PMAC and retrieve and place any pending responses within a response buffer for your use. This is an efficient and *safe* function to use. The word *safe* is emphasized because there are functions (i.e. PmacSendChar*X*(), PmacGetLineExX() and PmacSendLine*X*()) which, if not used carefully can cause un-synchronized communication (the mismatching of PMAC Commands to PMAC responses). PmacGetResponseEx*X*() always matches the command string with the response string or else it "times out".

Arguments

dwDevice	Device number.
response	Pointer to string buffer to copy the PMAC's response into.
maxchar	Maximum characters to copy.
command	Pointer to NULL terminated string to be sent to the PMAC as a question/command.

Return Value

If successful this function returns the number of characters received including handshake characters as well as the error code if any. Otherwise FALSE (0) which implies of course that an error occurred, or no characters were received since PMAC was not required to respond.

Error Codes

This routine initially sets the internal error flag to FALSE (0). Therefore calling PmacGetError() after this function will inform of any errors that occurred.

PmacGetControlResponseExA
PmacGetControlResponseExW

PmacGetControlResponseEx*X*()

```
int  PmacGetControlResponseExA(DWORD  dwDevice,  PCHAR
      response, UINT maxchar, CHAR ctl_char)

int  PmacGetControlResponseExW(DWORD  dwDevice,  PWCHAR
      response, UINT maxchar, WCHAR ctl_char)
```

Sends a control character to PMAC and returns a response, similar to PmacGetResponse*X*().

Arguments

dwDevice	Device number.
response	Pointer to string buffer. PMAC's response is placed there by function.
maxchar	Maximum characters to copy into response parameter
ctl_char	Control character to send to PMAC.

Return Value

Number of characters returned in response parameter and error code if any.

PmacDPRRealTimeEx

Purpose: Used to enable the DPR RealTime data buffer for PMAC Types only

Parameters:

on = TRUE for enable, FALSE for DISABLE
mask = bits determine which motors are updated in RAM (TURBO), I59 in non-turbo PMAC

Returns: Status of DPR Realtime data buffer, TRUE = ON, FALSE = OFF

PmacDPRRealTimeEx

```
                    BOOL PmacDPRRealTimeEx( DWORD dwDevice, long
                       mask, UINT period, int on )
{
  char str[20 ];
  if (vh[dwDevice].DPRAM == NULL)
    return FALSE;
  if (gh[dwDevice].bIsTurbo)
  {
    // Turbo
    if (on == TRUE)
    {
      wsprintfA( str, "i47=%u", period );
      PmacSendCommandA( dwDevice, str );
      PmacSendCommandA( dwDevice, "i48=1" );
      vh[dwDevice].DPRRTBufferTurbo->hostbusy = 0;  // Reset host busy
      vh[dwDevice].DPRRTBufferTurbo->dataready = 0; // Reset Data
Ready bit
      gh[dwDevice].DPRRealtActive = TRUE;
    }
    else
    {
      PmacSendCommandA( dwDevice, "i47=0" );
      PmacSendCommandA( dwDevice, "i48=0" );
      gh[dwDevice].DPRRealtActive = FALSE;
      vh[dwDevice].DPRRTBufferTurbo->dataready = 0; // Reset Data
Ready bit
    }
    vh[dwDevice].DPRRTBufferTurbo->motor_mask = mask;
  }
  else
  {
    if (on == TRUE)
    {
      wsprintfA( str, "i19=%u", period );
      PmacSendCommandA( dwDevice, str );
      PmacSendCommandA( dwDevice, "i48=1" );
      PmacSendCommandA( dwDevice, "gat" );    // turn on real time
buffer gather
      vh[dwDevice].DPRRTBuffer->hostbusy = 0; // Reset host busy
      gh[dwDevice].DPRRealtActive = TRUE;
    }
    else
    {
      PmacSendCommandA( dwDevice, "endgat" ); // turn off     real
time buffer gather
      PmacSendCommandA( dwDevice, "i48=0" );
      gh[dwDevice].DPRRealtActive = FALSE;
    }
```

```
  }
  return gh[dwDevice].DPRRealtActive;
}
```

PmacDPRBackgroundEx()

```
BOOL PmacDPRBackgroundEx(DWORD dwDevice, int on);
```

Starts or stops the PMAC's automatic write into the DPR background fixed and variable buffer.

Arguments

dwDevice	Device number.
On	Turn on or off Boolean.

Returns

TRUE if success.

PmacDPRVarBufInitEx()

```
long    PmacDPRVarBufInitEx(DWORD    dwDevice,    long
num_entries, long *addrarray);
```

Initializes a multi-user background variable buffer. (VBGDB)

User's VBGDB structure is allocated and updated with correct address offset and data offset. If other users are present, then their VBGDB structures will also be updated.

The buffer's start address and size will be initialized along with the address array in the dual ported RAM.

Assumptions

That this buffer was the last one to be initialized in the DPR buffer area.

That the DPR gather size is correct modulo of the number of variables and their size (32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48-bit PMAC variable. For example have 3 variables 2 x 24 and 1 x 48 which takes 4 x 32 bits. So the size must be an exact modulo of 4 because the PMAC dual ported RAM gather does not check before storing that there is enough room remaining.

That only one device structure is initialized per PMAC in the system

Arguments

dwDevice	Device number.
num_entries	The size for address Buffer.
addrarray[]	Array of address and types.

Return Value

A user handle if the Variable buffer is initialized OK otherwise 0.

PmacDPRVarBufChangeEx()

```
long PmacDPRVarBufChangeEx(DWORD dwDevice, long
     handle, long num_entries, long *addrarray)
```

Changes a previously initialized multi-user background variable buffer (VBGDB).

User's VBGDB structure is allocated and updated with correct address offset and data offset. If other users are present, then their VBGDB structures will also be updated.

The buffer's start address and size will be initialized along with the address array in the dual ported RAM.

Assumptions

- That this buffer was the last one to be initialized in the DPR buffer area.
- That the DPR gather size is correct modulo of the number of variables and their size (32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48-bit PMAC variable. For example have 3 variables 2 x 24 and 1 x 48 which takes 4 x 32 bits. So the size must be an exact modulo of 4 because the PMAC dual ported RAM gather does not check before storing that there is enough room remaining.
- That only one device structure is initialized per PMAC in the system

Arguments

dwDevice	Device number.
handle	Handle to previously initialized VBGDB.
num_entries	The size for address Buffer.
addrarray[]	Array of address and types.

Return Value

If it all works out a TRUE will be returned otherwise FALSE.

PmacDPRWriteBufferEx()

```
int     PmacDPRWriteBufferEx(DWORD     dwDevice,     int
    num_entries, struct VBGWFormat *the_data);
```

This feature is available for PROM Version 1.15G and above.

The variable background write buffer can be used to have PMAC transfer up to 32 PMAC long or short words from DPR to its own internal memory without using the DPR ASCII feature. The data to be written into PMAC's memory is first placed in an array of VBGWFormat structures (definition shown below).

```
struct VBGWFormat{
      long type_addr;
      long data1;
      long data2;
};
```

The upper 16 bits of type_addr element specifies the type of data and the lower 16 bits specify the address to which data1 and data2 should be written.

The types of data that may be specified are as follows.

Bit	Data
Bits 0-2	0/1/2/4 for Y/L/X/SPECIAL memory respectively. For SPECIAL (PROM V 1.16 and beyond) the user must modify bits B13 and B15 for the type (0 = PLCC Ladder Diagram Function Blocks, 1..7 are for future expansion).
Bits 3-7	Width = 1,4,8,12,16,20 (a value of 0 is 24 bit variable).
Bits 8-12	Offset = 0..23.
Bits 13-15	Special type

The way in which one uses data1, and data2 is not intuitive. For Y or X memory data1 element will be used for specifying the 24 bits of data. The most significant byte of data1 and all of data 2 are irrelevant in this case. For L data, PMAC 48 bit word, data1 should hold the first 32 bits and data2 should hold the most significant 16 bits leaving the most significant 16 bits of data2 irrelevant.

TWS is not yet implemented.

PMAC addresses are specified using an array of long integers. The most significant word of each long (upper 16 bits) specifies the word type. A value of 0, 1, 2 and 4 corresponds to Y, Long, X, and SPECIAL respectively. For Y, Long and X entries the least significant word specifies the actual PMAC address to be copied. For SPECIAL (PROM V 1.16 and beyond) the user must modify bits B13 and B15 for the type (0 = PLCC Ladder Diagram Function Blocks, 1..7 are for future expansion). When using the PLCC Ladder Diagram Function selection the least significant word specifies the function block structure member.

Arguments

num_entries Number of elements in the the_data array to be used, max 32

the_data A pointer to an array of VBGWFormat structures which specify what is copied from DPR into PMAC's internal memory.

entry_num Data entry to be received from table, see above for more detail

```
GetCurrentLabelEx
GetNumOfErrorsEx
AddErrRecordEx
PmacSetQuickGatherEx
PmacSetQuickGatherWithDirectCurrentEx
PmacDPRRealTimeEx
```

```
BOOL CALLBACK PmacDPRRealTimeEx(DWORD dwDevice, long mask, UINT
period, int on )
{
  char str[20 ];

  if (vh[dwDevice].DPRAM == NULL)
    return FALSE;

  if (gh[dwDevice].bIsTurbo)
  {
    // Turbo
    if (on == TRUE)
    {
      wsprintfA( str, "i47=%u", period );
```

```
      PmacSendCommandA( dwDevice, str );
      PmacSendCommandA( dwDevice, "i48=1" );
      vh[dwDevice].DPRRTBufferTurbo->hostbusy = 0;  // Reset host
busy
      vh[dwDevice].DPRRTBufferTurbo->dataready = 0; // Reset Data
Ready bit
      gh[dwDevice].DPRRealtActive = TRUE;
    }
    else
    {
      PmacSendCommandA( dwDevice, "i47=0" );
      PmacSendCommandA( dwDevice, "i48=0" );
      gh[dwDevice].DPRRealtActive = FALSE;
      vh[dwDevice].DPRRTBufferTurbo->dataready = 0; // Reset Data
Ready bit
    }
    vh[dwDevice].DPRRTBufferTurbo->motor_mask = mask;
  }
  else
  {
    if (on == TRUE)
    {
      wsprintfA( str, "i19=%u", period );
      PmacSendCommandA( dwDevice, str );
      PmacSendCommandA( dwDevice, "i48=1" );
      PmacSendCommandA( dwDevice, "gat" );    // turn on real
time buffer gather
      vh[dwDevice].DPRRTBuffer->hostbusy = 0; // Reset host busy
      gh[dwDevice].DPRRealtActive = TRUE;
    }
    else
    {
      PmacSendCommandA( dwDevice, "endgat" ); // turn off
real time buffer gather
      PmacSendCommandA( dwDevice, "i48=0" );
      gh[dwDevice].DPRRealtActive = FALSE;
    }
  }
  return gh[dwDevice].DPRRealtActive;
}
```

COMMUNICATION APPLICATION NOTES

Common Communication Traps

Last Updated: 10/30/2003, COMTraps.DOC

1. **Timeout!** Without a doubt, timeout is what communications is all about. Increasing the timeout has solved nearly 40-50% of all communication problems encountered by end users of PMAC. What is the timeout? It is a numeric value that is proportional to the time that the host computer waits for PMAC to respond (therefore it is a host computer parameter, not PMAC). If the host doesn't wait long enough, unsychronized communications occur (causing unpredictable behavior). Applications such as the PMAC Executive Program in most cases enable the end user to modify the timeout.

2. The second most experienced PMAC communication trap people run into is the "PMAC Output Buffer Blues" (**POBB**). Many times programmers are unaware of PMAC's two ASCII buffers, the input and output ASCII buffer, both of which are 255 characters long ("input" and "output" are used from the perspective of PMAC, i.e. input means flow from host to PMAC and output vice versa). Regardless of which method of communication is being used the following discussion is valid.

The symptoms of POBB are that motion programs and PLC's stall, and any further incoming ASCII commands are placed into the PMAC input buffer, but do not get processed. The cause is that there are two or more pending responses for the host in PMAC's output buffer. The remedy is to purge PMAC's output buffer, with a CONTROL-X and a stripping of any extra characters waiting on the bus or serial port. Once this is done, motion programs and PLC's and any queued up commands in PMAC's input buffer are processed. Many POBB victims claim that the remedy is plugging there stalled stand alone PMAC into a terminal (i.e. the executive program). But we know that this is just a side effect of the terminal purging PMAC's output queue.

There are many ways to get into POBB. But most are explained by the fact that people don't follow the following rule:

"Thou shall not talk to PMAC without listening"

That is, whenever a host sends PMAC an ASCII command string, the host should always check to see if PMAC has a response for the host (wait a timeout period). If I3 is set to 2 or 3 for example, PMAC will nearly always have something to say when the host sends it something. Another rule that if not followed will end up in a POBB condition is:

"Thou shall check PMAC periodically for unsolicited responses"

If the code in PMAC has run time errors (always caused by an illegal or badly timed "CMD" statement) PMAC will send an error code string out the active ASCII port. Also if the code in PMAC has any "SEND" commands there will be ASCII responses in PMAC 's output queue. If several of these happen and the host is not checking for these unsolicited responses, POBB will occur. For those stand alone applications out there, I1 may solve this problem by allowing PMAC to purge itself. But if you have a POBB problem, it behooves you to find out what error conditions exist at run time.

3. Initialization of communications is the third most often encountered problem. For those who are creating their own code, the following is a suggested procedure for initializing communication over the bus. Keep in mind that for DOS and Windows, Delta Tau does have libraries which take care of all this for you.

Suggested Startup Procedure for BUS

The following is a suggested initialization procedure for talking over the bus.

1. Write a value of 0 to I/O port BUSBASE + 5 and BUSBASE +6
2. Issue a control-x (ascii 24) to purge PMAC buffers
3. Wait 30 millisecods so control-x takes effect
4. Do a "GetLine()" several times to ensure a purged buffer. (Control-x removes all but two ascii characters, which are not accessable by PMAC)
5. Issue a test to confirm communications to PMAC has been established.
6. If test failed, increase timeout, issue a control-d to kill plc's and go back to step

Serial Communications

There have been reports that people have had to use the Executive Program before their own home brewed or 3rd party serial communications software would work. The sufficient conditions for this to occur are:

- PMAC has firmware 1.16 or before
- PMAC has just powered on or been reset
- The third party software uses hardware handshaking

From PMAC firmware version 1.16 and before, PMAC, after power up or a reset, would have it's RTS line set to FALSE, meaning it would hold off a host computer from sending PMAC commands. If the host computer would ignore the hardware handshaking for the first communication to PMAC, PMAC would then set it's RTS line to TRUE. Hardware handshaking could then be supported by the host computer. This explains why running the Executive program enabled other third party software (that uses hardware handshaking) to begin communicating. *The PMAC Executive Program does not look at the CTS line before sending PMAC commands (rather it looks at the UART's buffer empty bit instead). PMAC doesn't actively use the CTS line to hold off the host.*

Solutions: If you have firmware versions 1.16 and before and you are stuck using 3rd party software, you can easily solve this problem by shorting the host serial port's RTS and CTS lines (Pins 4 & 5 on PC's).

The following is a suggested initialization procedure for talking over the serial port:

1. Send out a Control-Z and then a Control-X to activate serial communications and purge any pending ASCII communications within PMAC.
2. Send to PMAC "I1=2@1 +" to check if PMAC's have been daisy chained
3. If PMAC reports an error (i.e. "^7ERR003") because of the "+" you have confirmed that there is more than one PMAC on the serial cable.
4. If PMAC does not report an error to the "+" command then there is only one PMAC on this serial line. Send "I1 = 0" to tell PMAC to use hardware handshaking and that it is not daisychained other PMAC's.

Slow Serial Communication

Subject: Correcting slow serial communication.

Applicable to: All PMAC's

Symptom: PMAC serial communication is slow.

Background:

PMAC has the ability to synchronize motors existing on different PMAC cards. When precise control is desired for motors existing on different cards, a single source for the phase and servo clock must be used. This can be accomplished by using one card as a master clock and having all other cards operate as slaves. The cards are chained together via an RS-232 cable. The phase clock and servo clock are routed

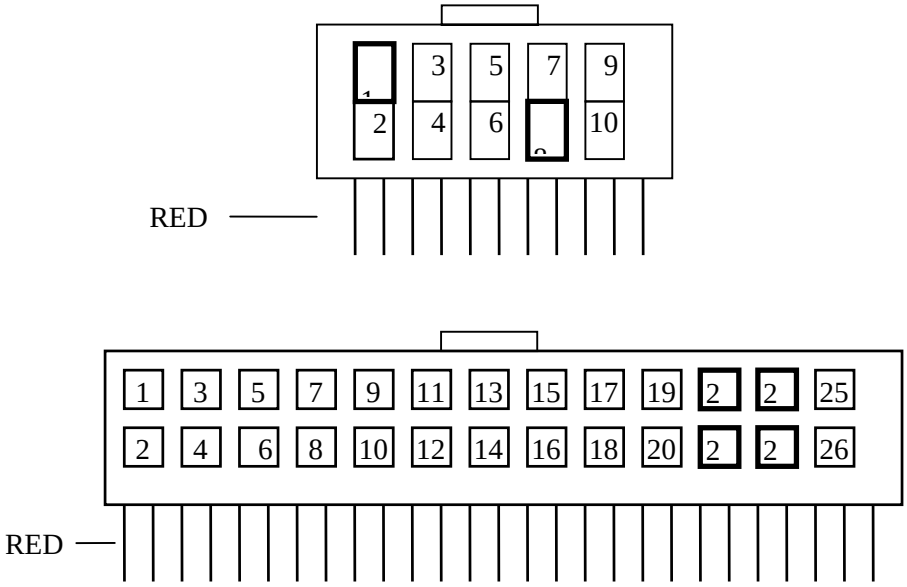
to pins on the serial cable. These signals may conflict with some operating systems (WINDOWS) use of these pins for serial communications to the PC or terminal.

On PMAC 26 pin connectors, the phase clock is routed to pin 23-24 and the servo clock is routed to pin 21-22.

On PMAC2 10-pin serial connectors, the phase clock is routed to pin 1 and the servo clock is routed to pin 8.

Solution:

For serial communication to a terminal program, it is suggested that the flat serial cable be altered by cutting these wires. As shown below:



Less Common Communication Problems

1. Trying to generate an interrupt on the PC via a PMAC PLC or motion program. The problem manifest itself by getting an interrupt for every toggling of the bit, but the interrupt vector comes out to be 0. The cause here is toggling the software bit too quickly (The bit that causes PMAC to generate the interrupt, see users manual). Put a small half or 1 millisecond delay between toggling the bits.
2. Can't communicate to a PMAC w/flash over the bus before running PMAC Executive first. Solution has found to be writing a 0 to the base address of the PMAC board.
3. Serial Communications programs derived from PMAC poll don't communicate until first running the PMAC Executive Program. Remove the line which checks the CTS in the sendline() routine, i.e.:
   a. While (i++ < timeout && (inportb (combase+5) & 32) == 0);
4. Can't establish communication serially. If a Control-T character (put PMAC in full duplex) was sent to PMAC, the application you are trying to use may not be anticipating this. Put E51 (or E3 on PMAC2) and try again.

Rare Communication Problems

1. When using a PMAC2 with DPR and attempting to use interrupt based communications, setting I56 to 1 causes the DPR to become all 1's. (For firmware versions 1.15G and before)
2. Control-X doesn't seem to effectively clear out PMAC's buffer. If a CTL_X is the last character in PMAC's input queue at the time PMAC checks it then all pending input and output characters in PMAC's queues get flushed. If a character gets sent after the CTL_X, making CTL_X not the last character in the queue, PMAC first parses the string. In the next background cycle PMAC will enforce the CTL_X. Because of this, even though a CTL_X was issued, if the host computer doesn't wait 100 ms or so (time depends on how busy PMAC is with higher priority tasks) PMAC could end up having characters for the host to strip.
3. Serial communications with daisy chained PMAC's. The problem is that when addressing a different card (i.e. "@2") sometimes the acknowledge character (or other requested data) is not received (if I3 is set such that an acknowledge character is expected, 2 or 3). This problem may occur because when issuing the command to switch from one card to the other, each card acts on the ASCII command in its background cycle. PMAC processes it's background cycle tasks in whatever time is left over from higher priority tasks. If the PMAC you are addressing has very little to do in it's high priority tasks (i.e. closing loops, motion program planning, PLC0 etc) and the currently active PMAC has a lot to do in it's high priority tasks, the addressed PMAC has a high probability of becoming the active PMAC before the the other PMAC became inactive. That is both PMAC's become active for a short period of time. When this happens data can be lost, or perhaps an unexpected character may be received. It is therefore wise to wait a few milliseconds after addressing a card before requesting data from PMAC.
4. Serial communications with daisy chained PMAC's with firmware version 1.16A and above.

The problem is that PMAC is not responding to the command sent to it, in fact it looks like it completely ignored it. It's easier to discuss this by considering an example. Let's say we've got two PMAC's, PMAC_A and PMAC_B. Both have firmware version 1.16A and are daisy chained together. And to make the situation as failure prone as possible we set the baud rates on both PMAC's to 38400. PMAC_A is very busy with motion programs and PLC's while PMAC_B is idle. Assume that PMAC_A is the currently addressed card and we wish to talk with PMAC_B and get some motor positions while we're at it. We therefore send the following command, "@1 #1p #2p #3p #4p #5p". You may find that the above command only partially get's to PMAC_B. This is because PMAC_A will take to much time in giving up the right to receive incoming commands from the host computer, and therefore some characters are lost.

To keep from losing commands in this situation the following procedure can be used:

1. Switch the currently addressed board to board number "x".
2. Query PMAC for the currently addressed board with the "@" command.
3. Do 2 until the appropriate board number comes up, "x" in this case.

Common Dual Ported Ram Traps

Last Updated: 10/30/2003, DPRTraps.DOC

This document contains a list of common Dual Ported Ram (DPR) troubles PMAC users may run into. If you know of or find additional items please e-mail to: ALLENS@DELTATAU.COM

Before the listing some fundamentals of DPR should be discussed. The basic steps in setting up the DPR are (See "PMAC Users Manual" and "PMAC Dual Ported Ram" for complete Detail):

1. Configure PMAC's DPR by writing to addresses. For VME users these addresses are from

a. X:$78A to X:$78C. For PC users the addresses are X:$786-X:$787. The Executive program should take care of this for you, but you might want to check it by hand by issuing a read command to PMAC (i.e. "rhx$783,10" or rhx$786,2").

2. Issue a save command.
3. Issue a $$$ command.
4. VME users additionally must write to the mailbox base plus 0x121 to (from the host computer) completely *setup the DPR address.*

The most common problem: DPR is in a location that conflicts with other devices or programs in the host computer's system. The solution here is to either use a system utility to find a free location for the DPR or trial and error (also see the application note "How to make sure your DPR is working via Brute Force Methods", DPRBRUTE.DOC). Furthermore, most often operating systems provide a means to reserve a memory block that you specify. Below a remedy for both DOS/Windows 3.x and Windows 95/98 and NT 4.0 are detailed.

DOS/Windows 3.x Users

DOS based users may use the MSD.EXE utility. You should also use the exclude switch with the EMM386.EXE entry of the CONFIG.SYS file to tell the operating system to reserve a spot in memory for the DPR. For example to exclude 16k of byte memory starting at physical address 0xD4000 the following entry in the config.sys should help:

```
DEVICE = EMM386.EXE /X D400-D7FF
```

Use the PMAC Executive Program to test if the DPR is in a good location. Usually the Configure Communications section has a test DPR feature.

Windows 95/98 & NT 4.0 Users

The "System Resources" dialog box enables you to view and reserve available memory. To get there click on the "Control Panel" icon and then click on the "System" icon. From there you should see a tabbed dialog box, click on the "Device Manager" tab. Here your computer resources are listed for you using a tree view hierarchy. Select the "Computer" icon and then press the "Properties" button. Yet another tabbed dialog comes up with two tabs, "View Resources", and "Reserve Resources". Use the "View" to scope out available memory, and the "Reserve" to claim your memory. For example to reserve physical address 000D4000-000D7FF0 click on the "Reserve Resources" tab and then press the "Add" button, then enter the appropriate values.

1. Shadow Ram has not been disabled in the computer BIOS. Shadow ram has been known to cause problems with using PMAC's DPR. Disable it by configuring your BIOS settings of your computer.
2. The contents of DPR are not what the user was expecting. PLC's may be writing to DPR or are indirectly affecting the contents/enabling of DPR. Also, check that all DPR automatic features (see DPR users manual) are disabled.
3. DPR becomes inaccessible after setting I56=1. This is a firmware glitch that has been fixed. This problem only occurred on PMAC2 for firmware versions 1.15G to 1.16A.
4. You don't have DPR. Whatever the reason, you might not have the DPR. For PMAC and PMAC-Lite the DPR is separate board that plugs in alongside the main PMAC board. For PMAC-VME and versions of PMAC2, the DPR is located on the main board of the controller as a socketed chip. PMAC-STD users simply can't have DPR.

5. DPR Communications is flaky, noisy, or simply isn't working like it did for the last 5 years. Check and make sure the cables connecting the DPR (if your DPR uses them) are well seated. Also make sure that the daughter board is well seated on the main board. Make sure the DPR chip is well seated in its socket if it is socketed.

Rare Problems

1. It appears as if there are two sets of ram, one the PMAC uses, the other the PC. It sounds like you haven't excluded the memory correctly. See the top of this article for setting this up.
2. DPR is fine on slow computers but not fast (P100's or faster). Or, from the host, it appears that every other byte is 0xFF. There was a modification in a programmable array logic chip for the DPR for PMAC2's and PMAC Ultralights. Please consult factory for details on changing this chip (XYLINX chip).

Terminal Example Program

This example program is a "CONSOLE" terminal application which illustrates a creation of a simple terminal. For serial communication, checksums can be turned on and off. This sample program links with RUNTIME.CPP to accomplish runtime linking.

```
#include "runtime.h"

#include <iostream.h>
#include <string.h>
#include <conio.h>

void show_printf (char *lbuf);

void main(){
  BOOL DONE,DEVICEISOPEN=false;
  BOOL  ChecksumEnabled;
  CHAR     output_buf[256]; // holds output string
  CHAR     input_buf[16384];  // holds input string
  CHAR  out_char;
  LONG  char_counter,temp;
  DWORD dwDevice;

      if((OpenRuntimeLink()) == NULL){
            printf("Error occurred in RuntimeLink\n");
            return;
      }

      printf("P32.EXE
\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1 Copyright DELTA TAU 1998\n\n");
  printf("\n       Created by Allen Segall, ALLENS@DELTATAU.COM\n");
  printf("  Use this simple terminal program for trouble shooting your
PMAC application.\n");
  printf("  NOTE: This application requires that the PComm32 driver be
installed.\n");

  printf("\n\n  Press CONTROL-I to initialize communication");
  printf("\n    If you have multiple PMAC devices, press CONTROL-I to
switch between them.");
  printf("\n  Press ESCAPE to shutdown communcation and exit.\n");
  char_counter=0;
  DONE = FALSE;
  ChecksumEnabled=FALSE;
  while(!kbhit() && !DONE)
```

```
  {
    switch(out_char = getch())
    {
      case 27: // Escape for EXIT
        DONE = true;
        break;
      case 9: // Control - I for "INITIALIZE"
        if(DEVICEISOPEN)
          DeviceClose(dwDevice);
        printf("\nWhat device number [0]?\n");
        gets(input_buf);
        dwDevice = atoi(input_buf);
        if(dwDevice <= 0)
          dwDevice = 0;
           if(!DeviceOpen(dwDevice))
        {
          printf("Device Open Failed\n");
          DEVICEISOPEN = false;
        }
        else
        {
          printf("Device Open Succeeded\n");
          printf("A terminal is active for your use.\n");
          DEVICEISOPEN = true;
        }
        break;
      case 20: // CTRL-T Checksummed communication ON
         if(ChecksumEnabled)
         {
           printf("Checksums Disabled\n");
           ChecksumEnabled = FALSE;
         }
         else
         {
           printf("Checksums Enabled\n");
           ChecksumEnabled = TRUE;
         }
         DeviceSERDoChecksums(dwDevice,ChecksumEnabled);
        break;
      case 13:
       putch(10);
        output_buf[char_counter] = '\x0';
        if(ChecksumEnabled)
        {
          temp =
DeviceSERCheckResponseA(dwDevice,input_buf,16384,output_buf);
          if(temp == 0)// An error occurred
          {
            switch(DeviceGetError(dwDevice))
            {
              case IDS_INCOMP_RECEIVE: // Unable to get complete
response
                printf("Did not receive the full response\n");
```

```
                break;
              case IDS_CHECKSUMDATABAD:
                printf("Data checksum error\n");
                break;
              case IDS_CHECKSUMACKBAD:
                printf("Acknowledge checksum error\n");
                break;
            }
          }
        }
        else
          DeviceGetResponse(dwDevice,input_buf,16384,output_buf);

        show_printf (input_buf);
        output_buf[0] = '\x0';
        char_counter = 0;
        break;
      default:
        output_buf[char_counter] = out_char;
        char_counter++;
        putch(out_char);
        break;
    }
  }

  if(DEVICEISOPEN)
    DeviceClose(dwDevice);

  CloseRuntimeLink();
}

void show_printf (char *lbuf)
{
      size_t          i;

      for (i = 0; i < strlen (lbuf); i++) {
            if (lbuf [i] == 13 /*||lbuf [i] == 7*/ || lbuf [i] == 6
||lbuf [i] == 10)
                  switch (lbuf [i]) {
                        case 13: printf ("<CR>\r\n"); break;
                        case 7:  printf ("<BELL>\r\n"); break;
                        case 6:  printf ("<ACK>"); break;
                        case 10: printf ("<LF>\r\n"); break;
                  }
            else {
                  if (isprint (lbuf [i]))
                        putch (lbuf [i]);
                  else
                        printf ("<%u>",lbuf [i]&255);
            }
      }
}
```

INDEX

B

Background Fixed Data Buffer 20
  Query Routines 60
Buffer
  Background Fixed Data 59
  Background Fixed Data Handshaking 59
  Background Fixed Data Initializing 59
  Data Query Function Description 23
  DPR Binary Rotary 15
  DPR Binary Rotary Motion Program 26
  DPR Fixed Background Data 13
  DPR Real Time Fixed Data 19
  DPR Real Time Fixed Data 55
  DPR Variable Background Read/Writer Data 22
  Fixed Real Time Data 13
  Memory Organization/Correct Order of Initialization 15
  Real Time Fixed Data Handshaking 55
  Real Time Fixed Data Initialization 55
  Variable Background Data 13
  Variable Read Data 22
  Variable Write Data 25
Buffer Query Routines
  Coordinate System 56
  Global 55
  Motor 56

C

Callbacks 95
ClosePmacDevice() 7, 32
Communication Application Notes 107
Communication Modes 5
    Bus 5
    Dual Ported Ram 5
    Serial Port 5
Communication Problems
  Less Common 109
  Rare 110
Communication Traps 107
Constants 95
Control Panel
  Dual Ported RAM 50

D

Data Query Function Description 23
Data Types, Structures, Callbacks, and Constants 95
Document Layout 1
DOS/Windows 3.x Users 111
DOWNLOADGETPROC 95
Downloading "Stuffing" Functions 27
Downloading ASCII PMAC Data 11
Downloading Multiple File Cooperative 11
DOWNLOADMSGPROC 95
DOWNLOADPROGRESS 95
DPR
  Action Bits 17
  Binary Rotary Buffer 15
  Control Panel 13, 17
  Features 13
  Fixed Background Data Buffer 13
  Fixed Real Time Data Buffer 13
  Initialization/Shutdown 15
  Numeric Read And Write 77
  Numeric Read And Write General Information 77
  Read/Write Functions 15
  Request Bits 17
  Support Functions 13
Dpr_Bin_Rot_Errors 96
DPRTESTMSGPROC 95
DPRTESTPROGRESS 96
Dual Ported Ram Traps 110
Dual Word Conversion 77

E

Error Handling 8

F

Feedrate Override
  Assigning New 18
  Enable Bits 18
Functions
  ASCII Communication 38
  Coordinate System Specific DPR Background Fixed Data Buffer 21
  Data Query
    Coordinate System 60
    Logical Query 61
    Motor Specific 60
  Data Query 60
  Data Query Coordinate System DPR Real Time Fixed Data Buffer 19
  Data Query DPR Background Fixed Data Buffer 21
  Data Query DPR Real Time Fixed Data Buffer 19

Data Query Global DPR Real Time Fixed Data Buffer 19
Data Query Motor DPR Real Time Fixed Data Buffer 19
Downloading 44
Dual Ported RaM Binary Rotary Buffer 69
Extended 99
Handshaking Background Fixed Data Buffer 20
Interrupt 83
Interrupt General Information 83
Logical Query DPR Background Fixed Data Buffer 21
Motor Specific DPR Background Fixed Data Buffer 21
Serial Checksum Communication 85
Startup/Shutdown DPR Real Time Fixed Data Buffer 19
Startup-Shutdown DPR Background Fixed Data Buffer 20
Utility 91
Variable 89

G

getBitValue() 92
GLOBALSTATUS 97

H

hex_long2() 93

I

InitDocument() 5
Initializing Communication 5
Installation 3
Fail 3
with Windows 95/98 3
with Windows NT 3
Interrupts 10
Call A Function 10
Send A Message To A Window 10
Set An Event 11
Start A Thread Function 10
Introduction 1

L

Library 2
Linking Your Code with the Driver 4

M

Masking 77
MOTION 98
MotionExe 6
MOTIONMODE 98

O

OpenPmacDevice() 32

P

PComm32 Features 10
PmacAbortDownload() 44
PmacAbortDPRTest() 32
PmacAddDownloadFile*X*() 44
PmacCompilePLCC*X*() 44
PmacConfigure() 33
PmacDownload() 45
PmacDownload() function 11
PmacDownloadFile() 45
PmacDownloadFile() function 11
PmacDownloadFirmwareFile() 46
PmacDPRAsciiFileToRot() 69
PmacDPRAsciiStrToBinaryFile() 69
PmacDPRAsciiStrToRot() 70
PmacDPRAsciiToBinaryFile() 71
PmacDPRBackground() 63
PmacDPRBackgroundEx() 102
PmacDPRBinaryToRot() 72
PmacDPRBufLast() 24, 27, 63
PmacDPRControlPanel() 50
PmacDPRControlPanel() function 17
PmacDPRDWordBitSet() 78
PmacDPRGet*Action*Bit() 50
PmacDPRGetDWord() 79
PmacDPRGetDWordMask() 79
PmacDPRGetFloat() 79
PmacDPRGetFOEnableBit() 51
PmacDPRGetFOValue() 51
PmacDPRGetMem() 79
PmacDPRGetRequestBit() 53
PmacDPRGetVBGAddress() 64
PmacDPRGetVBGAddrOffset() 64
PmacDPRGetVBGDataOffset() 64
PmacDPRGetVBGNumEntries() 64
PmacDPRGetVBGServoTimer() 65
PmacDPRGetVBGStartAddr() 24, 65
PmacDPRGetWord() 80
PmacDPRRealTimeEx 101
PmacDPRRotBuf() 27, 75

PmacDPRRotBufChange() 27, 74
PmacDPRRotBufClr() 27
PmacDPRRotBufInit() 26, 73
PmacDPRRotBufRemove() 27, 74
PmacDPRSet*Action*Bit() 50
PmacDPRSetDWord() 80
PmacDPRSetDWordBit() 80
PmacDPRSetDWordMask() 81
PmacDPRSetFloat() 81
PmacDPRSetFOEnableBit() 51
PmacDPRSetFOValue() 51
PmacDPRSetMem() 81
PmacDPRSetRequestBit() 53
PmacDPRSetWord() 82
PmacDPRTest() 33
PmacDPRVarBufChange() 65
PmacDPRVarBufInit() 66
PmacDPRVarBufInitEx() 102
PmacDPRVarBufRead() 66
PmacDPRVarBufRemove() 67
PmacDPRWriteBuffer() 67
PmacDPRWriteBufferEx() 103
PmacFlush() 38
PmacGetAsciiComm() 38
PmacGetBuffer*X*() 38
PmacGetControlResponseEx*X*() 100
PmacGetControlResponse*X*() 39
PmacGetDpramAvailable() 33
PmacGetIORegister() 92
PmacGetIVariable() 90
PmacGetLineEx*X*() 99
PmacGetLine*X*() 39
PmacGetPmacType() 91
PmacGetResponseEx*X*() 100
PmacGetRomDateA() 91
PmacGetRomVersionA() 91
PmacGetUserHandle() 91
PmacGetVariable() 89
PmacInBootStrapMode() 33
PmacINTRFireEventInit() 83
PmacINTRFuncCallInit() 84
PmacINTRRunThreadInit() 84
PmacINTRTerminate() 10, 84
PmacINTRWndMsgInit() 84
PmacMultiDownload() 46
PmacReadReady() 40
PmacRemoveDownloadFile*X*() 47
PmacSendChar*X*() 41
PmacSendCommand*X*() 41
PmacSendLine*X*() 41
PmacSERCheckGetLineA() 86
PmacSERCheckResponse() Source/Example 86
PmacSERCheckResponseA() 85
PmacSERCheckSendLineA() 86
PmacSERDoChecksums() 85
PmacSERGetBaudrate() 34
PmacSERGetHandle() 35
PmacSERGetPort() 35
PmacSERIsOpen() 34
PmacSEROnLine() 35
PmacSERSetBaudrate() 35
PmacSERSetComm() 36
PmacSERSetPort() 36
PmacSetAsciiComm() 41
PmacSetIORegister() 92
PmacSetIVariable() 89
PmacSetMaxDownloadErrors() 48
PmacSetVariable() 89
PMACTEST program 4
PmacThreadDownload*X*() 48
PROGRAM 98
Programmers Reference 30

Q

Query Routines
DPR Real Time Fixed Data Buffer 55
Quick Guide to ASCII Communication Functions 7

S

SendMessage() function 10
Serial Communication 108
Serial Communications 108
SERVOSTATUS 98
Shutting Down Communication 7
Standard Read/Write 77
Startup Procedure for Bus 108
Structures 95

T

Terminal Example Program 112

U

ulMask parameter 10
UNICODE Versus ANSI Compiles 8
Usage of PComm32 4

V

Variable Background Data Buffer 13
Variable Read Data Buffer

Startup/Shutdown 22

W

Windows 95/98 & NT 4.0 Users 111